

Steffen Bangsow

# Tecnomatix Plant Simulation

Modeling  
and  
Programming  
by Means  
of Examples



Springer

# Tecnomatix Plant Simulation

Steffen Bangsow

# Tecnomatix Plant Simulation

Modeling and Programming by Means  
of Examples



Springer

Steffen Bangsow  
Freiligrathstrasse 23  
08058 Zwickau  
Germany  
steffen@bangsow.net

Translated by Steffen Bangsow

ISBN 978-3-319-19502-5      ISBN 978-3-319-19503-2 (eBook)  
DOI 10.1007/978-3-319-19503-2

Library of Congress Control Number: 2015940995

Springer Cham Heidelberg New York Dordrecht London  
© Springer International Publishing Switzerland 2015

Translation from the German language edition: *Praxishandbuch Plant Simulation and Simtalk* by Steffen Bangsow, © Carl Hanser Verlag, Munich 2011. All rights reserved

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media  
([www.springer.com](http://www.springer.com))

# Preface

Based on the competition of international production networks, the pressure to increase the efficiency of production systems has increased significantly. In addition, the number of technical components in many products and as a consequence also the requirements for corresponding assembly processes and logistics processes increases. International logistics networks require corresponding logistics concepts.

These requirements can be managed only by using appropriate Digital Factory tools in the context of a product lifecycle management environment, which allows reusing data, supports an effective cooperation between different departments, and provides up-to-date and relevant data to every user who needs it.

Simulating the complete material flow including all relevant production, storage, and transport activities is recognized as a key component of the Digital Factory in the industry and as of today widely used and accepted. Cutting inventory and throughput time by 20–60% and enhancing the productivity of existing production facilities by 15–20% can be achieved in real-life projects.

The purpose of running simulations varies from strategic to tactical up to operational goals. From a strategic point of view, users answer questions like which factory in which country suits best to produce the next generation product taking into account factors like consequences for logistics, worker efficiency, downtimes, flexibility, storage costs, etc., looking at production strategies for the next years. In this context, users also evaluate the flexibility of the production system, e.g., for significant changes of production numbers — a topic which becomes more and more important. On a tactical level, simulation is executed for a time frame of 1–3 months in average to analyze required resources, optimize the sequence of orders, and lot sizes. For simulation on an operational level, data are imported about the current status of production equipment and the status of work in progress to execute a forward simulation till the end of the current shift. In this case, the purpose is to check if the target output for the shift will be reached and to evaluate emergency strategies in case of disruptions or capacities being not available unexpectedly.

In any case, users run simulation to take a decision about a new production system or evaluate an existing production system. Usually, the value of those systems is a significant factor for the company, so the users have to be sure that they take the right decision based on accurate numbers. There are several random processes in real production systems like technical availabilities, arrival times of assembly parts, process times of human activities, etc., so stochastic processes play an important role for throughput simulation. Therefore, Plant Simulation provides a whole range of

easy-to-use tools to analyze models with stochastic processes, to calculate distributions for sample values, to manage simulation experiments, and to determine optimized system parameters.

Besides that, results of a simulation model depend on the quality of the input data and the accuracy of the model compared to the behavior of the real production system. As soon as assembly processes are involved, several transport systems with their transport controls, workers with multiple qualification profiles or storage logic, production processes become highly complex. Plant Simulation provides all necessary functionality to model, analyze, and maintain large and complex systems in an efficient way. Key features like object orientation and inheritance allow users to develop, exchange/reuse, and maintain their own objects and libraries to increase modeling efficiency. The unique Plant Simulation optimization capabilities support users to optimize multiple system parameters at once like the number of transporters, monorail carriers, buffer/storage capacities, etc., taking into account multiple evaluation criteria like reduced stock, increased utilization, increased throughput, etc.

Based on these accurate modeling capabilities and statistic analysis capabilities, typically an accuracy of at least 99% of the throughput values is achieved with Plant Simulation models in real-life projects depending on the level of detail. Based on the price of production equipment, a return on investment of the costs to introduce simulation is quite often already achieved after the first simulation project.

Visualizing the complete model in the Plant Simulation 3D environment allows an impressive 3D presentation of the system behavior. Logfiles can be used to visualize the simulation in a Virtual Reality (VR) environment. The support of a Siemens PLM Software unified 3D graphics engine and unified graphics format allows a common look-and-feel and easy access to 3D graphics which were created in other tools like digital product design or 3D factory layout design tools.

The modeling of complex logic always requires the usage of a programming language. Plant Simulation simplifies the need to work with programming language tremendously by supporting the user with templates, with an extensive examples collection and a professional debugging environment.

Compared to other simulation tools in the market, Plant Simulation supports a very flexible way of working with the model, e.g., by changing system parameters while the simulation is running.

This book provides the first comprehensive introduction to Plant Simulation. It supports new users of the software to get started quickly, provides an excellent introduction how to work with the embedded programming language SimTalk, and even helps advanced users with examples of typical modeling tasks. The book focuses on the basic knowledge required to execute simulation projects with Plant Simulation, which is an excellent starting point for real-life projects.

We wish you a lot of success with Tecnomatix Plant Simulation.

November 2009

Dirk Molfenter †  
*Siemens PLM Software*

# Contents

<b>1</b>	<b>Basics .....</b>	<b>1</b>
1.1	Introducing Material Flow and Logistics Simulation.....	1
1.1.1	Uses .....	1
1.1.2	Definitions .....	2
1.1.3	Procedure of Simulation .....	2
1.1.3.1	Formulation of Problems.....	3
1.1.3.2	Test of Simulation-Worthiness.....	3
1.1.3.3	Formulation of Targets.....	3
1.1.3.4	Data Collection.....	3
1.1.3.5	Modeling .....	4
1.1.3.6	Executing Simulation Runs .....	5
1.1.3.7	Result Analysis and Result Interpretation .....	5
1.1.3.8	Documentation .....	6
1.2	Plant Simulation: First Steps .....	6
1.2.1	The Tutorial.....	6
1.2.2	Step-by-Step Help .....	6
1.2.3	Example Collections and Demo Videos.....	7
1.2.4	The Siemens PLM Software Community.....	7
1.3	Introductory Example.....	8
1.4	First Simulation Example .....	9
1.4.1	Insert Objects into the Frame .....	9
1.4.2	Connect the Objects .....	10
1.4.3	Define the Settings of the Objects .....	10
1.4.4	Run the Simulation.....	11
1.5	Modeling .....	11
1.5.1	Object-Related Modeling .....	11
1.5.2	Object-Oriented Modeling .....	12
1.6	Student and Demo Version .....	14
<b>2</b>	<b>SimTalk and Dialogs .....</b>	<b>17</b>
2.1	The Object Method .....	17
2.2	The Method Editor .....	19
2.2.1	Line Numbers, Entering Text.....	19
2.2.2	Bookmarks .....	19

2.2.3	Code Completion.....	20
2.2.4	Information about Attributes and Methods .....	20
2.2.5	Templates .....	22
2.2.6	The Debugger.....	22
2.3	SimTalk.....	23
2.3.1	Names.....	23
2.3.2	Anonymous Identifiers .....	24
2.3.3	Paths .....	25
2.3.3.1	Absolute Path .....	25
2.3.3.2	Relative Path .....	25
2.3.3.3	Name Scope .....	25
2.3.4	Comments .....	26
2.4	Variables and Data Types .....	27
2.4.1	Variables .....	27
2.4.2	Local Variables .....	27
2.4.3	Arrays.....	29
2.4.4	Global Variables.....	30
2.5	Operators.....	32
2.5.1	Mathematical Operators .....	32
2.5.2	Logical (Relational) Operators.....	33
2.5.3	Assignments .....	33
2.6	Branching .....	35
2.7	Case Differentiation .....	37
2.8	Loops.....	38
2.8.1	Conditional Loops .....	38
2.8.1.1	Header-Controlled Loops.....	38
2.8.1.2	Footer-Controlled Loops.....	39
2.8.2	For Loop.....	39
2.9	Methods and Functions .....	41
2.9.1	Passing Arguments.....	41
2.9.2	Passing Several Arguments at the Same Time .....	42
2.9.3	Result of a Function .....	43
2.9.4	Predefined SimTalk Functions .....	45
2.9.5	Method Call.....	47
2.9.5.1	Sensors .....	47
2.9.5.2	Other Events for Calling Methods.....	50
2.9.5.3	Constructors and Destructors .....	52
2.9.5.4	Drag and Drop Control.....	53
2.9.5.5	Method Call after a Certain Timeout .....	53
2.9.5.6	Recursive Programming.....	55
2.9.5.7	Observer .....	57
2.10	Interrupt Methods.....	58
2.10.1	The Wait Statement.....	58
2.10.2	Suspending of Methods .....	58

2.11	Debugging, Optimization .....	61
2.11.1	Breakpoints in Methods .....	61
2.11.2	Breakpoints in the EventController .....	64
2.11.3	Error Handler .....	64
2.11.4	Profiler .....	66
2.12	Hierarchical Modeling .....	69
2.12.1	The Frame .....	69
2.12.2	The Interface .....	69
2.12.3	Create Your Own Libraries .....	72
2.13	Dynamic Model Generation .....	73
2.13.1	Required Data, SimTalk Language Elements .....	73
2.13.2	Create Objects and Set Attributes .....	74
2.12.3	Link Objects Dynamically with Connectors .....	76
2.13.4	Connect Objects Dynamically with Lines .....	77
2.14	Dialogs .....	79
2.14.1	Elements of the Dialog .....	79
2.14.1.1	The Dialog Object .....	80
2.14.1.2	Callback Function .....	82
2.14.1.3	The Static Text Box .....	83
2.14.1.4	The Edit Text Box .....	83
2.14.1.5	Images in Dialogs .....	84
2.14.1.6	Buttons .....	85
2.14.1.7	Radio Buttons .....	87
2.14.1.8	Checkbox .....	88
2.14.1.9	Drop-Down List Box and List Box .....	89
2.14.1.10	List View .....	90
2.14.1.11	Tab Control .....	93
2.14.1.12	Group Box .....	93
2.14.1.13	Menu and Menu Item .....	93
2.14.2	Accessing Dialogs .....	94
2.14.3	User Interface Controls .....	95
2.14.4	Input Functions .....	96
2.14.5	Output Functions .....	98
2.14.5.1	Messagebox .....	98
2.14.5.2	Infobox .....	100
2.14.5.3	Bell and Beep .....	100
2.14.5.4	HTML Window .....	100
<b>3</b>	<b>Modeling of Production Processes .....</b>	<b>103</b>
3.1	Material Flow Library Elements .....	103
3.1.1	General Behavior of the Material Flow Elements .....	103
3.1.1.1	Time Consumption .....	104
3.1.1.2	Capacity .....	106

3.1.1.3	Blocking, Exit Behavior .....	106
3.1.1.4	Failures .....	108
3.1.2	ShiftCalendar .....	111
3.2	SimTalk Attributes and Methods of the Material Flow Elements ...	118
3.2.1	States of the Material Flow Elements .....	118
3.2.2	Setup .....	126
3.2.3	Finished Messages .....	128
3.2.3.1	Finished Messages Using <code>resWorking</code> .....	128
3.2.3.2	Create Your Own Finished Messages .....	129
3.2.4	Content of the Material Flow Objects .....	132
3.3	Mobile Units (MUs) .....	132
3.3.1	Standard Methods of Mobile Units .....	132
3.3.2	Length, Width and Booking Point .....	134
3.3.3	Entity and Container .....	136
3.4	Source and Drain .....	138
3.4.1	Basic Behavior of the Source .....	138
3.4.2	Settings of the Source .....	139
3.4.3	Source Control Using a Trigger .....	143
3.4.4	User-defined Source with SimTalk .....	146
3.4.5	The Drain .....	148
3.5	Single Processing .....	148
3.5.1	SingleProc, Fixed Chained Machines .....	149
3.5.2	Batch Processing .....	149
3.6	Simultaneous Processing of Several Parts .....	154
3.6.1	The ParallelProc .....	154
3.6.2	Machine with Parallel Processing Stations .....	157
3.6.3	Continuous Machining, Fixed Transfer Lines .....	159
3.6.4	The Cycle, Flexible Cycle Lines .....	160
3.7	Assembly Processes .....	162
3.7.1	The AssemblyStation .....	162
3.7.2	Assembly with Variable Assembly Tables .....	164
3.7.3	Use SimTalk to Model Assembly Processes .....	166
3.8	Dismantling .....	169
3.8.1	The DismantleStation .....	169
3.8.2	Simulation of Split-Up Processes .....	173
3.8.3	Dismantle Processes Using SimTalk .....	174
3.9	Scrap and Rework .....	176
3.9.1	The FlowControl .....	176
3.9.2	Model Scrap Using the Exit Strategy .....	178
<b>4</b>	<b>Information Flow, Controls .....</b>	<b>181</b>
4.1	The List Editor .....	181
4.2	One-Dimensional Lists .....	182

4.2.1	The CardFile.....	182
4.2.2	StackFile and QueueFile .....	183
4.2.3	Searching in Lists .....	184
4.3	The TableFile .....	184
4.3.1	Methods and Attributes of the TableFile.....	185
4.3.2	Searching in TableFiles .....	186
4.3.3	Calculating within Tables.....	190
4.3.4	Nested Tables and Nested Lists.....	191
4.4	TimeSequence .....	193
4.4.1	The Object TimeSequence .....	193
4.4.2	TimeSequence with TableFile and SimTalk .....	196
4.5	The Trigger .....	197
4.5.1	The Object Trigger .....	197
4.5.2	Trigger with SimTalk and TableFile .....	200
4.6	The Generator .....	206
4.6.1	The Generator Object.....	206
4.6.2	User-defined Generator with SimTalk .....	207
4.7	The AttributeExplorer .....	209
4.8	The EventController.....	211
4.9	Shop Floor Control, Push Control.....	216
4.9.1	Base Model Machine .....	217
4.9.2	Elements of the Job Shop Simulation.....	218
4.9.2.1	Work Plans .....	219
4.9.2.2	Order Management.....	221
4.9.2.3	Resource Management .....	224
4.9.2.4	Production Control .....	225
4.10	Pull Control .....	227
4.10.1	Simple Pull Control .....	227
4.10.2	Kanban .....	228
4.10.2.1	Functioning of the Kanban System .....	228
4.10.2.2	Control Loops .....	229
4.10.2.3	Modeling of a Single-Stage E-Kanban System .....	229
4.10.2.4	Bin Kanban System.....	239
4.10.2.5	Card Kanban System.....	243
4.10.3	The Plant Simulation Kanban Library .....	243
4.11	Line Production.....	246
4.11.1	CONWIP Control .....	246
4.11.2	Overall System Availability, Line Down Time .....	249
4.11.3	Sequence Stability .....	253
<b>5</b>	<b>Working with Random Values .....</b>	<b>261</b>
5.1	Working with Distribution Tables.....	261
5.2	Working with Probability Distributions .....	267
5.2.1	Use of DataFit to Determine Probability Distributions .....	267

5.2.2	Use of Uniform Distributions.....	270
5.2.3	Set of Random Distributed Values Using SimTalk.....	270
5.3	Warm-Up Time .....	271
5.4	The ExperimentManager.....	272
5.4.1	Simple Experiments .....	273
5.4.2	Multi-level Experimental Design .....	275
5.5	Generic Algorithms.....	277
5.5.1	GA Sequence Tasks .....	277
5.5.2	GA Range Allocation.....	279
<b>6</b>	<b>Simulation of Transport Processes .....</b>	<b>283</b>
6.1	The Line .....	283
6.1.1	Attributes of the Line .....	283
6.1.2	Curves and Corners .....	285
6.2	AngularConverter and Turntable.....	286
6.2.1	Settings of the AngularConverter.....	288
6.2.2	Settings of the Turntable .....	288
6.2.3	Turntable, Select User-defined Exit .....	289
6.3	The Turnplate .....	290
6.3.1	Basic Behavior of the Turnplate.....	290
6.3.2	Settings of the Turnplate .....	290
6.4	The Converter .....	292
6.5	The Track .....	295
6.6	Sensors on Length-Oriented Blocks.....	296
6.6.1	Function and Use of Sensors .....	296
6.6.2	Light Barrier Mode.....	299
6.6.3	Create Sensors Automatically .....	301
6.7	The Transporter.....	303
6.7.1	Attributes of the Transporter .....	303
6.7.2	Load and Unload the Transporter Using the AssemblyStation and the DismantlingStation .....	305
6.7.3	Load and Unload the Transporter Using the TransferStation .....	307
6.7.4	Load and Unload Transporter Using SimTalk .....	309
6.7.5	SimTalk Methods and Attributes of the Transporter.....	309
6.7.6	Stopping and Continuing.....	310
6.7.7	Drive a Certain Distance .....	314
6.7.8	Routing .....	319
6.7.8.1	Automatic Routing .....	319
6.7.8.2	Routing (Destination Lists) .....	321
6.7.8.3	Routing with SimTalk .....	324
6.7.8.4	Driving Control (“freestyle”) .....	325
6.7.9	Sensorposition, Sensor-ID, Direction.....	328
6.7.10	Start Delay Duration.....	332
6.7.11	Load Bay Type Line, Cross-Sliding Car .....	337

6.8	Tractor.....	342
6.8.1	General Behavior.....	342
6.8.2	Hitch Wagons to the Tractor .....	342
6.8.3	Loading and Unloading of Trains.....	345
6.9	Model Transporters with Battery .....	349
6.10	Case studies.....	353
6.10.1	The Plant Simulation Multi-Portal Crane Object .....	353
6.10.2	Simulation of a Forklift .....	357
<b>7</b>	<b>Simulation of Robots and Handling Equipment.....</b>	<b>363</b>
7.1	PickAndPlace .....	363
7.1.1	Attributes of the PickAndPlace Object.....	364
7.1.2	Blocking Angle .....	366
7.1.3	Time Factor .....	367
7.2	Simulation of Robots.....	368
7.2.1	Exit Strategy Cyclic Sequence .....	368
7.2.2	Load and Unload of Machines (Single Gripper) .....	369
7.2.3	Load and Unload of Machines (Double Gripper).....	370
7.2.4	PickAndPlace Loads Containers .....	372
7.2.5	Assembly with Robots .....	373
7.2.6	The Target Control of the PickAndPlace Object.....	375
7.2.7	Consider Custom Transport and Processing Times.....	377
7.2.8	Advantages and Limitations of the PickAndPlace Object .....	380
7.3	Model Handling Robots Using Transporter and Track .....	380
7.3.1	Basic Model and General Control .....	380
7.3.2	Partial Parameterized Control Development .....	386
7.3.3	Handling and Processing Times of the Robot .....	389
7.3.4	Synchronous and Asynchronous Control of the Robot .....	397
7.4	The LockoutZone .....	403
7.5	Gantry Robots .....	405
<b>8</b>	<b>Warehousing and Procurement .....</b>	<b>411</b>
8.1	Buffer .....	411
8.2	The Sorter.....	412
8.2.1	Basic Behavior .....	412
8.2.2	Attributes of the Sorter .....	412
8.2.3	Sort by Method.....	415
8.3	The Store, Warehousing .....	416
8.3.1	The Store .....	417
8.3.2	Chaotic Warehousing .....	417
8.3.2.1	Inventory, Process of Storage.....	419
8.3.2.2	Structure of the Inventory (Table Stock).....	419
8.3.2.3	Looking for a Free Place .....	419

8.3.2.4	Store and Register Parts .....	421
8.3.2.5	Find a Part and Remove It from the Warehouse.....	422
8.3.3	Virtual Warehousing .....	423
8.3.4	Extension of the Store Class.....	428
8.3.4.1	Search a Free Place, Store, Update Stock List ....	429
8.3.4.2	Search for and Retrieval of Parts.....	431
8.3.4.3	Stock Statistics .....	432
8.3.5	Simplified Warehousing Model .....	434
8.3.6	Warehouse Key Figures .....	439
8.3.7	Storage Costs.....	443
8.3.8	Economic Order Quantity .....	443
8.3.9	Cumulative Quantities.....	445
8.4	Procurement .....	446
8.4.1	Warehousing Strategies .....	447
8.4.2	Consumption-Based Inventory Replenishment .....	447
8.4.2.1	Order Rhythm Method .....	447
8.4.2.2	Reorder Point Method .....	455
8.4.2.3	Goods Receipt Warehouse, Reorder Point Method.....	461
	8.4.2.3.1 Warehouse Retrievals	470
	8.4.2.3.2 Deliveries.....	475
	8.4.2.3.3 Inbound, Out-of-stock Part .....	477
	8.4.2.3.4 Visualization of the Database Stock ....	481
	8.4.2.3.5 Storage and Retrieval Orders .....	483
	8.4.2.3.6 Warehouse Statistics (Database).....	485
8.4.3	The StorageCrane Object .....	486
8.4.3.1	Store and Remove Automatically with the StorageCrane .....	486
8.4.3.2	Customized Storage and Retrieval Strategies.....	488
8.4.3.3	Stock Statistics of the StorageCrane Object.....	492
8.4.3.4	Load and Unload the Store with a Transporter ....	494
<b>9</b>	<b>Simulation of Workers .....</b>	<b>499</b>
9.1	Exporter, Importer and Broker .....	499
9.1.1	Function.....	499
9.1.2	Exporter and Broker Statistics.....	501
9.2	Worker .....	501
9.2.1	The Worker-WorkerPool-Workplace-FootPath Concept ....	502
9.2.2	The Broker .....	503
9.2.3	The WorkerPool .....	503
9.2.4	The Worker .....	505
9.2.5	The Footpath .....	506
9.2.6	The Workplace .....	506
9.2.7	Worker Transporting Parts .....	507

9.3	Worker Statistics .....	509
9.4	Case Snippets for Worker Simulation .....	509
9.4.1	Loading of Multiple Machines by One Operator .....	509
9.4.2	The Worker Loads and Unloads Containers.....	511
9.4.3	Chaku-Chaku.....	514
9.4.4	Troubleshooting Depending on the Nature of the Failure ...	516
9.4.5	Collaborative Work of Several Workers .....	518
9.4.6	A Worker Executes Different Activities at One Station.....	521
9.4.7	The Worker Changes Speed Depending on the Load.....	523
9.4.8	Employees Working with Different Efficiency .....	524
9.4.9	The Worker Loads Carriers and Transports Them .....	525
9.4.10	Multiple-Machine Operation.....	527
9.4.11	Worker Loads Machines on Availability .....	529
9.4.12	Worker Works with Priority (Broker Importer Request Control) .....	531
9.4.13	Determination of the Number of Workers with the ExperimentManager .....	533
9.5	Modeling of Workers with Transporter and Track.....	534
9.5.1	Modeling Approach.....	534
9.5.2	The Worker Follows a Process.....	534
9.5.3	The Worker Is Driving a Transporter.....	538
<b>10</b>	<b>The Fluids Library .....</b>	<b>543</b>
10.1	The Fluid Elements, Continuous Simulation.....	543
10.2	The Tank .....	546
10.3	Simple Case Studies .....	548
10.3.1	Pumping Out .....	548
10.3.2	Distribute Fluids.....	551
10.3.3	Fill the Tank with a Tanker .....	552
10.3.4	Unload a Tank Using a Tanker.....	556
10.3.5	Separator .....	558
10.3.6	Status Change.....	560
<b>11</b>	<b>2D and 3D Visualization .....</b>	<b>563</b>
11.1	2D Visualization.....	563
11.1.1	The Icon Editor .....	563
11.1.2	Inserting Images .....	564
11.1.2.1	Insert Images from the Clipboard.....	564
11.1.2.2	Inserting Images from a File .....	564
11.1.2.3	Changing the Background of the Frame.....	565
11.1.3	Animation Structures and Reference Points .....	566
11.1.3.1	Set Reference Points .....	566
11.1.3.2	Animation Structures .....	568
11.1.4	Animating Frames .....	570

11.1.5	Dynamic Creation of 2D Animation Structures .....	572
11.1.6	Simple 2D Icon Animations .....	576
11.2	Plant Simulation 3D .....	578
11.2.1	Introduction to Plant Simulation 3D.....	578
11.2.2	Navigation in the 3D Scene.....	579
11.2.3	Formatting 3D Objects .....	579
11.2.3.1	Load 3D Graphics .....	580
11.3.2.2	Transformations .....	581
11.3.2.3	Animation Paths 3D .....	582
11.2.4	3D State Icons (LEDs) .....	584
11.2.5	MU Animation .....	585
11.2.6	Length-Oriented Objects in 3D .....	586
11.2.7	Textured Plate .....	588
11.3	3D Animation.....	590
11.3.1	Self-animations, Named Animations.....	590
11.3.2	SimTalk 3D Animations, Unnamed Animations.....	591
11.3.3	Camera Animations.....	592
11.3.3.1	Attach Camera.....	592
11.3.3.2	Camera Path Animations.....	593
11.3.4	Manipulation of 3D Objects .....	594
<b>12</b>	<b>Integrate Energy Consumption and Costs .....</b>	<b>599</b>
12.1	Simulation of Energy Consumption .....	599
12.1.1	Energy Consumption—Basic Behavior .....	599
12.1.2	Energy Profiles.....	602
12.1.3	Energy Consumption in the Periphery of Machines.....	604
12.2	Integrate Costs into the Simulation .....	606
12.2.1	Production Concurrent Costing .....	606
12.2.2	Working Assets .....	608
12.2.3	Machine-Hour Rates .....	610
<b>13</b>	<b>Statistics .....</b>	<b>615</b>
13.1	Statistics Collection Period .....	615
13.2	Statistics—Methods and Attributes.....	617
13.2.1	Write the Statistical Data into a File.....	618
13.2.2	Determining Average Values .....	620
13.2.3	Record Values .....	621
13.2.4	Calculation of the Number of Jobs Executed Per Hour (JPH).....	622
13.2.5	Data Collected by the Drain .....	624
13.2.6	Statistical Values of the Global Variable .....	625
13.2.7	Statistics of the Transporter.....	626

13.3	User Interface Objects.....	632
13.3.1	The Chart Object .....	632
13.3.1.1	Plotter.....	633
13.3.1.2	Chart.....	636
13.3.1.3	Statistics Wizard.....	639
13.3.1.4	Histogram.....	640
13.3.2	The Sankey Diagram.....	641
13.3.3	The BottleneckAnalyzer.....	644
13.3.4	The Display .....	645
13.3.5	The Display Panel .....	646
13.3.6	The Comment.....	649
13.4	The Report.....	650
13.4.1	Automatic Resource Report (Statistics Report).....	650
13.4.2	The Report Until V. 11.....	650
13.4.3	The HTML Report (V. 12).....	657
<b>14</b>	<b>Data Exchange and Interfaces.....</b>	<b>661</b>
14.1	DDE with Plant Simulation.....	661
14.1.1	Read Plant Simulation Data in Microsoft Excel.....	661
14.1.2	Excel DDE Data Import in Plant Simulation.....	662
14.1.3	Plant Simulation DDE Data Export to Excel.....	664
14.1.4	Plant Simulation Remote Control.....	665
14.1.5	DDE Hotlinks.....	666
14.2	The COM Interface .....	667
14.2.1	Read Data from Plant Simulation.....	667
14.2.2	Write Data, Call Methods, Respond to Events .....	669
14.3	The ActiveX Interface .....	672
14.3.1	ActiveX and Excel .....	672
14.3.2	ActiveX Data Objects (ADO, ADOX) .....	677
14.3.2.1	Creating a New Access Database.....	677
14.3.2.2	Integrating the Necessary Libraries.....	678
14.3.2.3	Creating a Database Table.....	678
14.3.2.4	Insert Data into a Table, Add Records .....	679
14.3.2.5	Find and Display Records .....	680
14.3.2.6	Updating Data .....	681
14.4	The File Interface .....	682
14.5	The ODBC Interface .....	684
14.5.1	Setup an ODBC Data Source .....	684
14.5.2	Read Data from a Database .....	686
14.5.3	Write Data into a Database.....	688
14.5.4	Delete Data in a Database Table .....	689
14.5.5	SQL Commands .....	689

14.6	SQLite Interface .....	691
14.6.1	Create Databases and Tables.....	691
14.6.2	Working with SQLite Databases.....	693
14.6.3	SQL Functions in SQLite .....	697
14.7	The XML Interface .....	697
14.7.1	Introduction in XML .....	697
14.7.2	Read in XML Files into Tables .....	699
14.7.3	The XML Interface.....	699
14.7.3.1	Select Nodes and Read Values.....	700
14.7.3.2	Changing Data in XML Files .....	703
	<b>Subject Index.....</b>	<b>705</b>

# Chapter 1

## Basics

Simulation technology is an important tool for planning, implementing, and operating complex technical systems.

Several trends in the economy lead to shorter planning cycles. These include

- increasing product complexity and variety
- increasing quality demands in connection with high cost pressure
- increasing demands regarding flexibility
- shorter product life cycles
- shrinking lot sizes
- increasing competitive pressure

Simulation is used where simpler methods no longer provide useful results.

### 1.1 Introducing Material Flow and Logistics Simulation

#### 1.1.1 Uses

You can use simulation during the planning, implementation, and operation of equipment. Possible questions can include the following:

- Planning phase
  - Identification of bottlenecks in the derivation of potential improvement
  - Uncovering hidden, unused potentials
  - Minimum and maximum utilization
  - Juxtaposition of different planning alternatives
  - Testing of arguments regarding capacity, effectiveness of control, performance limits, bottlenecks, throughput speed, and volume of stocks
  - Visualization of planning alternatives for decision making
- Implementation phase
  - Performance tests
  - Problem analysis, performance test on future requirements
  - Simulation of exceptional system conditions and accidents
  - Training new employees (e.g. incident management)
  - Simulation of ramp-up and cool-down behaviors

- Operational phase
  - Testing of control alternatives
  - Review of emergency strategies and accident programs
  - Proof of quality assurance and fault management
  - Dispatching of orders and determination of the probable delivery dates

### 1.1.2 *Definitions*

#### **Simulation** (source: VDI 3633<sup>1</sup>)

Simulation is the reproduction of a real system and its dynamic processes in a model. The aim is to achieve transferable findings for reality. In a wider sense, simulation means preparing, implementing and evaluating specific experiments using a simulation model.

#### **System:** (VDI 3633)

A system is defined as a separate set of components that are related to each other.

#### **Model:**

A model is a simplified replica of a planned or real system with its processes in another system. It differs from the original in important properties only within specified tolerance levels.

#### **Simulation Run:** (source: VDI 3633)

A simulation run is the image of the behavior of the system in the simulation model within a specified period.

#### **Experiment:** (source: VDI 3633)

An experiment is a targeted empirical study of the behavior of a model through repeated simulation runs with a systematic variation of arguments.

### 1.1.3 *Procedure of Simulation*

According to VDI guideline 3633, the following approach is recommended:

1. Formulation of problems
2. Test of simulation-worthiness
3. Formulation of targets
4. Data collection and data analysis
5. Modeling
6. Execution of simulation runs
7. Result analysis and result interpretation
8. Documentation

---

<sup>1</sup> VDI-Richtlinie 3633 Blatt 1(2000), Simulation von Logistik-, Materialfluss- und Produktionssystemen, Grundlagen. VDI-Handbuch Materialfluss und Fördertechnik, Bd. 8, Gründruck Beuth, Berlin, 2000.

### 1.1.3.1 Formulation of Problems

Together with the customer of the simulation, the simulation expert must formulate the requirements for the simulation. The result of the formulated problem should be a written agreement (e.g. a technical specification), which contains concrete problems that will be studied using simulation.

### 1.1.3.2 Test of Simulation-Worthiness

To assess simulation-worthiness you can, for example, examine:

- The lack of analytical mathematical models (for instance, many variables)
- High complexity, many factors to be considered
- Inaccurate data
- Gradual exploration of system limits
- Repeated use of the simulation model

### 1.1.3.3 Formulation of Targets

Each company aims at a system of targets. It usually consists of a top target (such as profitability), which splits into a variety of sub-targets that interact with each other. The definition of the target system is an important preparatory step. Frequent targets for simulations are, for example:

- Minimize processing time
- Maximize utilization
- Minimize inventory
- Increase in-time delivery

All defined targets must be collected and analyzed statistically at the end of the simulation runs, which implies a certain required level of detail for the simulation model. Hence, they determine the range of the simulation study.

### 1.1.3.4 Data Collection

The data required for the simulation study can be structured as follows:

- System load data
- Organizational data
- Technical data

The following overview is a small selection of data to be collected:

**Table 1.1** Data Collection

Technical data	
Factory structural data	Layout Means of production Transport functions Transport routes Areas Restrictions
Manufacturing data	Use time Performance data Capacity
Material flow data	Topology Conveyors Capacities
Accident data	Functional accidents Availability
Organizational data	
Working time organization	Break scheme Shift scheme
Resource allocation	Worker Machines Conveyors
Organization	Strategy Restrictions Incident management
System load data	
Product data	Working plans Bill of materials
Job data	Production orders Transportation orders Volumes Dates

### 1.1.3.5 Modeling

The modeling phase includes building and testing the simulation model.

Modeling usually consists of two stages:

- 1 Derive an iconic model from the conceptual model
- 2 Transfer the model into a software model

#### First Modeling Stage

First, you must develop a general understanding of the simulated system. Depending on the objectives to be tested, you have to make decisions about the

accuracy of the simulation. Based on the accuracy of the simulation, necessary decisions are taken about which aspects to simplify. The first modeling stage covers two activities:

- Analysis (breakdown)
- Abstraction (generalization)

Using the system analysis, the complexity of the system in accordance with the original investigation targets will be dissolved through meaningful dissection of the system into its components. Using abstraction, the amount of the specific system attributes will be decreased as far as it is practical to form a limited image of the original system. Typical methods of abstraction are reduction (elimination of irrelevant details) and generalization (simplification of essential details).

### **Second Modeling Stage**

A simulation model will be built and tested. The result of modeling must be included in the model documentation to make further changes in the simulation model possible. In practice, this step is often neglected; hence, models cannot be used due to the lack of documentation of functionality. Therefore, there is a need for commenting on the models and the source code during programming. This ensures that the explanation of the functionality is still available after programming is complete.

#### **1.1.3.6 Executing Simulation Runs**

Depending on the objectives of the simulation study, the experiments based on a test plan will be realized. In the test plan, the individual experiments on output data, arguments of the model, objectives, and expected results are determined. It is also important to define a time span for the simulation experiments, based on the findings of the test runs. Computer runs spanning several hours or frequent repetitive experiments for statistical coverage are not uncommon. In these cases, it is helpful to check whether it is possible to control the experiments using a separate programmed object (batch runs). The realization times for the experiments can be relocated partly at night, so that the available computing capacity can be utilized optimally. Input and output data as well as the underlying parameters of the simulation model must be documented for each experiment.

#### **1.1.3.7 Result Analysis and Result Interpretation**

The values, which will change in the modeled system, are derived from the simulation results. The correct interpretation of the simulation results significantly influences the success of a simulation study. If the results contradict the assumptions made, it is necessary to analyze what influences are responsible for the unexpected results. It is also important to realize that complex systems often have a ramp-up phase. This phase may run differently in reality and in the simulation. Therefore, the results obtained during the ramp-up phase are often not transferable to the modeled system and may have no influence on the evaluation (Exception: The ramp-up phase of the original system has to be fully modeled).

### 1.1.3.8 Documentation

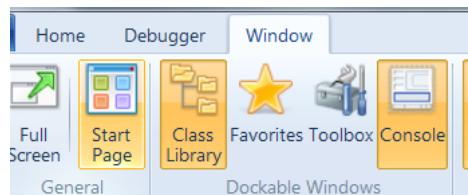
For the documentation of a simulation study, the form of a project report is recommended. The documentation should provide an overview of the timing of the study and document the work conducted. Of interest in this context is the documentation of failed system variants and constellations. The core of the project report should be a presentation of the simulation results based on the customer requirement specification. Resulting from the simulation study, it makes sense to include proposals for actions in the documentation. Finally, we recommend describing the simulation model in terms of its structure and functionality.

## 1.2 Plant Simulation: First Steps

If you are new to Plant Simulation, you should start with the material provided by Plant Simulation. As a first step, you should reproduce all examples from the Plant Simulation Tutorial.

### 1.2.1 The Tutorial

You gain access to the Tutorial via the Start Page of Plant Simulation. Until version 11, select View—Start Page; from version 12 onwards, click Window—Start Page (Fig. 1.1).



**Fig. 1.1** Start Page

Click on Tutorial. The online tutorial offers a quick start and guides you systematically in creating a simple simulation model. As a first step, you should practice all examples from the Plant Simulation Tutorial. After completing the tutorial, you will be ready to work with the examples in this book.

### 1.2.2 Step-by-Step Help

The Step-by-Step Help feature provides descriptions of steps that are necessary to model several tasks. It is part of the chapter on online help entitled “Step-by-Step Help” (Fig. 1.2). Note: In version 12, the Help feature is located in the File menu.



Fig. 1.2 Step-by-Step Help

The context-sensitive help in the dialogs of objects provides additional explanations of the dialog elements. To show context-sensitive help, click on the question mark on the top right corner of the dialog, and then click on the dialog element. The window of the context-sensitive help shows a reference to the corresponding SimTalk attribute at its end (professional license).

### 1.2.3 Example Collections and Demo Videos

Plant Simulation contains a variety of examples of small models that are thematically ordered and show how and with which settings you can use the components and functions. You find a link to the examples on the Start Page of Plant Simulation (Fig. 1.3). You also find some demo videos here.



Fig. 1.3 Example Collection

Most of the examples in this book (and many more) can be downloaded from my website: <http://www.bangsow.de>.

### 1.2.4 The Siemens PLM Software Community

If you have questions regarding Plant Simulation, you can visit the PLM Software Community (Fig. 1.4). Just follow the link from the Start Page of Plant Simulation (starting from version 12), or type this address in your browser: <http://community.plm.automation.siemens.com/>.



Fig. 1.4 Siemens PLM Software Community

### 1.3 Introductory Example

Start Plant Simulation by clicking on the icon in the program group or the desktop icon.

#### The Program Window (v. 12)

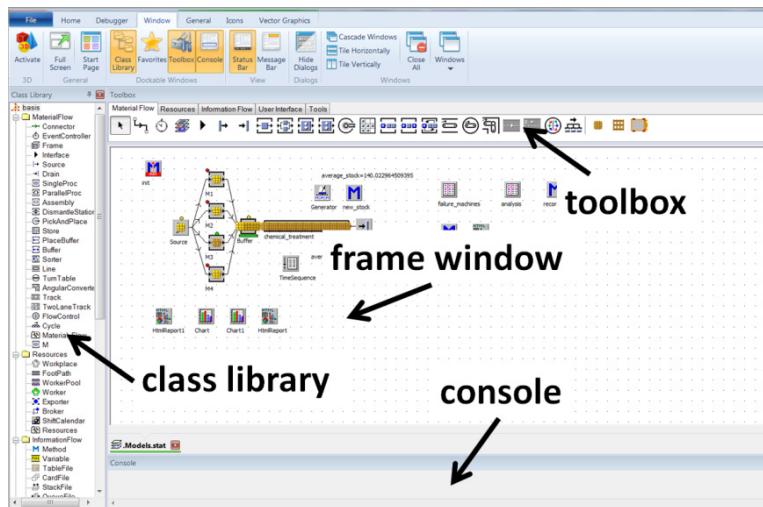


Fig. 1.5 Program Window

To define the layout, you can use the menu item: WINDOW. Here, you set what you want to see on the screen. A standard Plant Simulation window can, for example, contain the elements from Fig. 1.5.

You show and hide elements from the program in the Window menu (Fig. 1.6).



**Fig. 1.6** Window menu

### The Class Library

In the class library, you find all objects required for the simulation. You can create your own folders, derive and duplicate classes, create frames, or load objects from other simulation models.

### The Console

The console provides information during the simulation (e.g. error messages). You can use the Print command to output messages to the console. If you do not need the console, you can hide it.

### The Toolbox

The toolbox provides quick access to the classes in the class library. You can easily create your own tabs in the toolbox and fill it with your own objects.

## 1.4 First Simulation Example

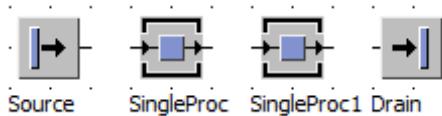
As a first example, a simple production line is to be built with a source (material producer), two workstations, and a drain (material consumer). Start Plant Simulation and select the menu command File—New. This opens a frame (window). Simulation models are created in the frame object.

### 1.4.1 Insert Objects into the Frame

To insert objects into the frame, you have two options:

- Click on the object icon in the toolbox, and then click in the frame. The object will be inserted into the frame at the position at which you clicked.
- Another way is to drag the object from the class library to the frame and drop it there (drag and drop).

Insert objects into the frame as in Fig. 1.7.



**Fig. 1.7** Model

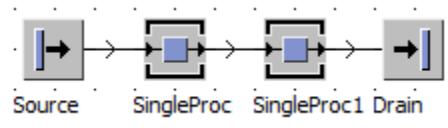
### 1.4.2 Connect the Objects

You have to connect the objects along the material flow, so that the different parts can be transported from one object to the next. This is what the connector object does. The Connector has an icon as depicted in Fig. 1.8 in the toolbar.



**Fig. 1.8** Connector icon

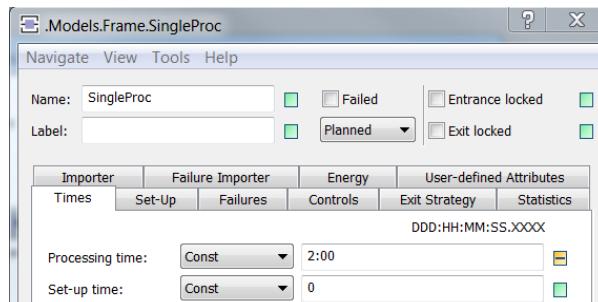
Click the connector in the toolbar, then the object in the frame that you want to connect (the cursor changed its icon on connector), and click on the next object. If you want to insert several connectors successively, hold down the CTRL key. The result should look like in Fig. 1.9.



**Fig. 1.9** Connected objects

### 1.4.3 Define the Settings of the Objects

You have to define some settings in the objects such as processing times, capacity, information for setup, failures, breaks, etc. Properties can be easily set in the dialogs of the objects. You can open a dialog by double-clicking an object. For example, set the following values: SingleProc stations: processing time: two minutes (Fig. 1.10); drain: processing time: zero seconds; source: two-minute interval.



**Fig. 1.10** Dialog window

The Apply button saves the values, but the dialog remains open. OK saves the values and closes the dialog. Finally, you need the event controller. It coordinates the processes that run during a simulation. By default, Plant Simulation inserts an EventController into the frame.

#### 1.4.4 Run the Simulation

You can control the simulation with the icons in the left, upper corner of the Home menu (Fig. 1.11).

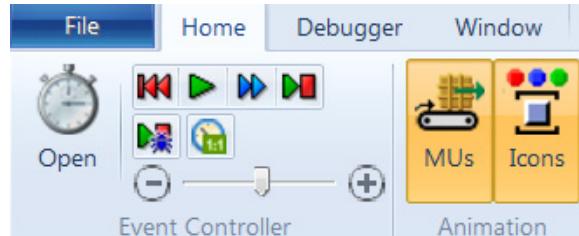


Fig. 1.11 EventController Buttons

Click the Start button to start the simulation and Stop to stop the simulation. In the frame, the material movements are graphically displayed (Fig. 1.12). You can now change the model to see what happens. In the Statistics tab of the objects you will find statistical values that are collected by Plant Simulation.

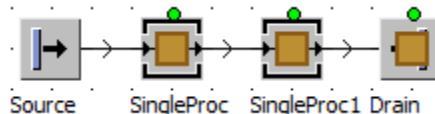
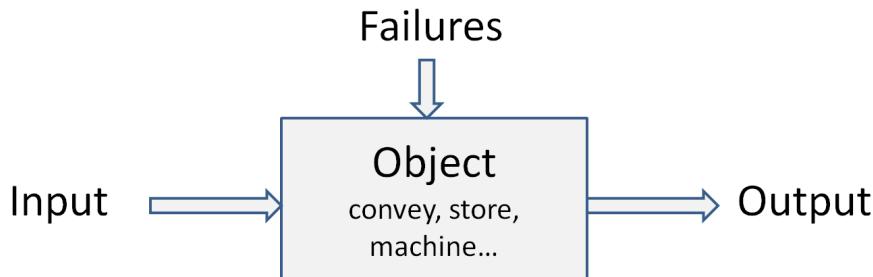


Fig. 1.12 Animated MUS

## 1.5 Modeling

### 1.5.1 Object-Related Modeling

In general, only a limited selection of objects is available for representing the real world. They are named, e.g. as model blocks, which show the real system with all the properties to be investigated. Hierarchically structured system models are best designed top-down. In this way, the real system will be decomposed into separate functional units (subsystems). If you are not able to model in a sufficiently precise manner using the available model objects, you should continue to decompose, etc. Each object must be described precisely (Fig. 1.13).



**Fig. 1.13** Simulation objects

The individual objects and the operations within the objects are linked to an overall process. This creates a frame. Using the objects and the frame, various logistical systems can be modeled.

### 1.5.2 *Object-Oriented Modeling*

#### Objects and Properties

The hierarchical structure of an object allows being addressed exactly (analogous to a file path). A robot Rob1 may be addressed in the hierarchy (the levels are separated by a period) as follows:

`production1.press_hall.section1.cell1.Rob1`

Rob1 is described by a number of properties, such as type of handling, speed, capacity, lead times, etc. All properties that describe Rob1 are called object. An object is identified by its name (Rob1) and its path:

`(production1.press_hall.section1.cell1.Rob1)`

The properties are called attributes. They consist of a property description (attribute type)—e.g. engine type—and a property value (attribute value)—e.g. HANUK-ZsR1234578.

#### Classes and Instances

In object-oriented programming, a class is defined as follows: A class is a user-defined data type. It designs a new data type to create a definition of a concept that has no direct counterpart in the fundamental data types.

Example: You want to create a new type of transport unit that cannot be defined by standard types. All definitions (properties, methods, behavior) required for creating a new type, are called a class. The individual manifestation of the class is called an instance of the class (e.g. Transport—Forklift [general]; Instance: Forklift 12/345 [concrete]). The instance has the same basic properties as the class and some special characteristics (such as a specific name).

#### Inheritance

In Plant Simulation, you can create a new class based on an existing class (derive a class, create a subclass). The original class is called base class and the derived

class is called subclass. You can expand a data type through the derivation of a class without having to redefine it. You can use the basic objects of the class by employing inheritance.

Example: You have several machines of the same type; hence, most of the properties are the same. Instead of defining each machine individually, you can define a basic machine. All other machines are derived from this basic machine. The subclasses inherited the properties of the base class they apply to these classes as if they were defined there.

### Duplication and Derivation

Try the following example: Select the SingleProc in the class library. Click the right mouse button to open the context menu. Select Duplicate from the Context menu. Plant Simulation names the duplicate SingleProc1. Change the processing time in the class SingleProc to two minutes. Open the dialog of the SingleProc1. The processing time has not changed.

The duplicate contains all the attributes of the original, but there is no connection between the original and the duplicate (there is no inheritance). You can also create duplicates using the mouse: Press the Control key and drag the object to its destination, then drop it.

Now do the same with Derive. Select the SingleProc again, click the right mouse button and click on Derive from the Context menu. Plant Simulation names the new class SingleProc2. With Derive you created an instance of the class. This instance can either be a new class (in the library) or an object (e.g. in a frame object). Initially, the instance inherits all the characteristics of the original class. Now, change the processing time of the SingleProc class to 10 minutes (10:00). Save the changes in the SingleProc and open the dialog of SingleProc2. SingleProc2 has applied the change in the processing time of the SingleProc class.

You can also derive in the class library using CTRL + SHIFT and dragging the mouse. You can navigate to the original class from an object or from a derived class. Double-click the class/the object and then select Navigate—Open Origin. You can also find a button on the Home menu (Fig. 1.14).

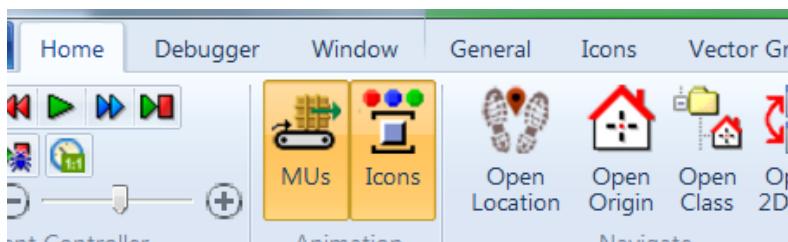


Fig. 1.14 Home Menu

This will open the dialog of the original class. If you drag a class from the class library into a frame object, the new object is derived. Try this: Open a frame object. Add a SingleProc to the frame object (via drag and drop from the library).

Change the processing time of the SingleProc in the library and check the processing time in the frame. The values are inherited from the object in the library.

### Duplicating Objects in the Frame

To duplicate an object in the frame, hold down the Ctrl key and drag the object to a free spot on the frame (the mouse pointer shows a “+”).

Using the object, you can also activate inheritance to the initial class.

Test: Change the processing time of the SingleProc in the library; then the processing time of both SingleProcs in the frame also changes. If you change the processing time of one SingleProc in the frame, the processing time of the other SingleProc in the frame will not be changed (the duplicate has no inheritance relationship with its original, but with the original class of the original). In the object dialogs, you can easily identify which values are inherited and which have been entered in the instance. Each attribute shows a green toggle button to the right. This is green if the value is inherited and yellow with a minus sign inside if the values are not the same as in the original class.

Value inherited	Value changed (inserted)
1:00	5:00

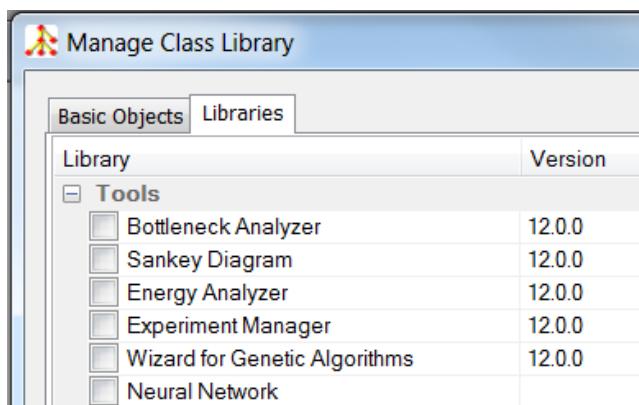
To restore the inheritance of a value, click the button after the value and then click Apply.

## 1.6 Student and Demo Version

The student and demo versions are restricted to 80 elements. In this context, method blocks, connectors and some other elements are not counted. To find out how many of the restricted elements are included in your model, proceed as follows: Add a method block into your frame. Enter the following code:

```
is
do
  print numOfLimitedObjects;
end;
```

The console displays the number of limited elements. In version 12, the model already contains after its creation 33 limited elements. This is the content of the Tools folder. You can remove any items you do not need from the model. In version 12, select Home—Model—Manage Class Library. Uncheck all items in the Libraries tab (Fig. 1.15).



The screenshot shows a software window titled "Manage Class Library". At the top, there are two tabs: "Basic Objects" and "Libraries", with "Libraries" being the active tab. Below the tabs is a table with two columns: "Library" and "Version". The table contains six rows, each representing a tool: "Tools", "Bottleneck Analyzer", "Sankey Diagram", "Energy Analyzer", "Experiment Manager", "Wizard for Genetic Algorithms", and "Neural Network". All tools are listed under the "Tools" category, and their version is 12.0.0.

Library	Version
Tools	
Bottleneck Analyzer	12.0.0
Sankey Diagram	12.0.0
Energy Analyzer	12.0.0
Experiment Manager	12.0.0
Wizard for Genetic Algorithms	12.0.0
Neural Network	

**Fig. 1.15** Manage Class Library

If you then check the number of elements, the result should be zero.

# Chapter 2

## SimTalk and Dialogs

The basic behavior of Plant Simulation objects in practice is often insufficient for generating realistic system models. To extend the standard features of the objects, Plant Simulation provides the programming language SimTalk. This allows modifying the basic behavior of individual objects. SimTalk can be divided into two parts: Control structures and language constructs (conditions, loops, etc.).

Standard methods of the material and information flow objects. These are built-in and form the basic functionality that you can use. You develop SimTalk programs in an instance of the information flow object Method.

### 2.1 The Object Method

You can create controls with Method objects, which are then called and started from the basic objects using their names. You find the Method in the class library within the folder InformationFlow (Fig. 2.1).



Fig. 2.1 Class Library

#### Example: Stock Removal

We want to simulate a small production using a store. The capacity of the store is 100 parts. SingleProc produces one part every minute (the source delivers). Create a frame according to Fig. 2.2. Name the frame "storage."

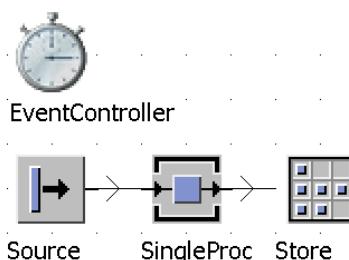
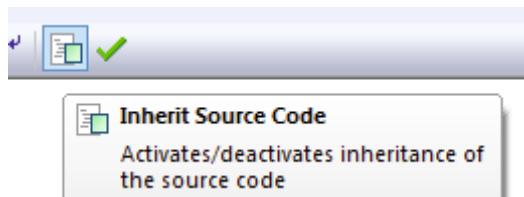


Fig. 2.2 Example Frame

If you now run the simulation, it soon results in a jam (storage is full). This is logical since there is no inventory decrease. You need a method that simulates an inventory decrease. Drag a method object to the frame. Double-clicking the icon opens the method. The methods (functions) always have the same structure:

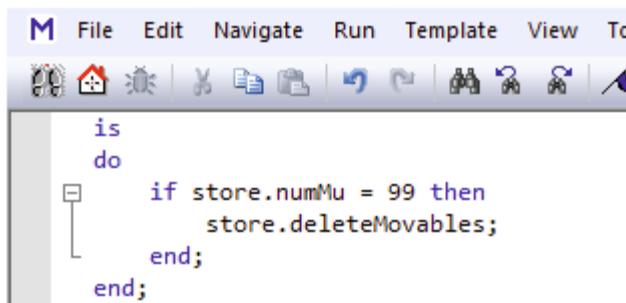
```
is
do
  -- Statements
end;
```

Declare variables between "is and do." Enter your source code between "do and end." First, turn off the inheritance. Click on the icon (Fig. 2.3) in the editor or select Tools—Inherit Source Code.



**Fig. 2.3** Inheritance

Formulate the instructions in SimTalk (call your method "stockRemoval"). Create the content according to Fig. 2.4.



**Fig. 2.4** Method Editor

Confirm your changes with F7 or and assign the method to an object. For this purpose, each object has one or more sensors. When an MU (Movable Unit) passes through a sensor, the relevant method is triggered. Double-click on Store—Controls—Entrance (Fig. 2.5); select the correct method and you are ready to proceed.

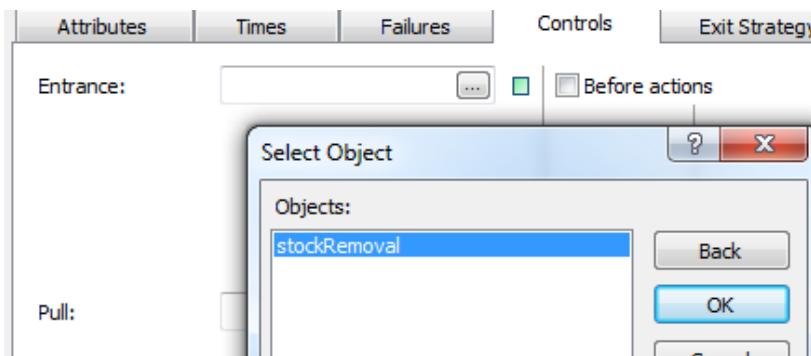


Fig. 2.5 Entrance Control

Now start the simulation. If you were successful, there would be no jam. The store it will verify the quantity for each entry. If it is 99, then the store will be emptied (a simple solution).

## 2.2 The Method Editor

Double-clicking a method object opens an editor. You will find a number of functions in the editor that facilitate your work during programming. If you cannot enter your source code into the method editor, inheritance is still turned on (see above).

### 2.2.1 *Line Numbers, Entering Text*

You can display line numbers using the command: View—Display Line Numbers. The following rules apply for entering text:

- Double-clicking selects a word
- Clicking three times selects a row
- Ctrl + A selects everything
- Copy does not work with a right click (until version 9). Use Ctrl + C to copy and Ctrl + V to insert text or use the menu commands Edit—Copy, etc.
- Use Ctrl + Z to undo the last change (or Edit Undo)
- Move also works by dragging with the mouse

### 2.2.2 *Bookmarks*

For faster navigation, set bookmarks in your code. The bookmarks are displayed in red (see also Table 2.1). To insert a bookmark, select any text and click: .

**Table 2.1** Bookmark functions

Icon	Description
	Deletes all bookmarks in the method
	Cursor moves to previous bookmark
	Cursor moves to next bookmark

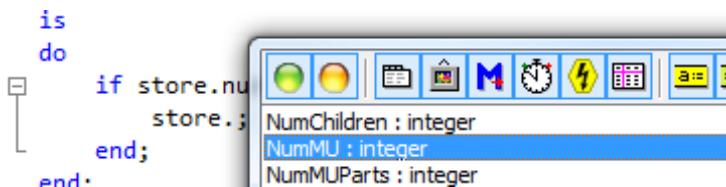
### 2.2.3 *Code Completion*

The editor supports automatic code completion. If there is only one possibility of completion, Plant Simulation shows the attribute, the method or variable as a light blue label (Fig. 2.6). You can accept the suggestion with **Ctrl + space bar**.

```
store.numMu = 99 t deleteMovable
  store.deleteMovable;
t;
```

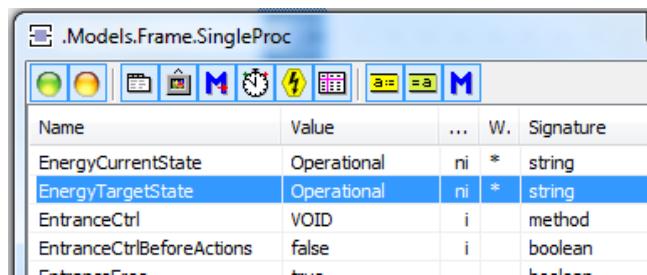
**Fig. 2.6** Code Completion

Starting from an object, you can display all possible completions. Simply press **CTRL + SPACE** (Fig. 2.7). In the list you can scroll using the direction buttons; an entry will be accepted with **Enter**.

**Fig. 2.7** Object Attributes and Methods

### 2.2.4 *Information about Attributes and Methods*

You can always get information about the built-in attributes and methods of an object through **Show Attributes and Methods** in the context menu of an object. In the table, you can see all methods and attributes (even those you have defined, Fig. 2.8) of the object.



Name	Value	...	W.	Signature
EnergyCurrentState	Operational	ni	*	string
EnergyTargetState	Operational	ni	*	string
EntranceCtrl	VOID	i		method
EntranceCtrlBeforeActions	false	i		boolean

**Fig. 2.8** Attributes and Methods

The column signature allows you to deduce whether it is a method or an attribute and which data you need to pass or what type is returned. If the column signature shows only the data type, then the entry is an attribute.

Example: Show the attributes and methods of the store. Look for RecoveryTime (Fig. 2.9).



receptive				
RecoveryTime	0.0000	ni		(MU:object) : boolean

**Fig. 2.9** Attribute

Recovery time is an attribute; the data is not in parentheses. In order to set the recovery time, you have to type:

Store.recoveryTime:=120;

Set the value of an attribute with “:=”.



PauseCtrl	VOID			method
PE		ni		(X:integer, Y:integer) : any

**Fig. 2.10** Method

PE is a method (Fig. 2.10). It expects two arguments of data-type integer and returns any possible type (usually an object). PE allows you to access a particular place of the store. Call the method using parentheses:

Store.PE(1,1);

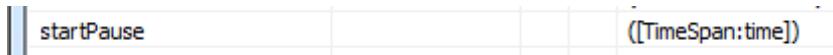


mirrorX	VOID			method
mirrorY		ni		(X:integer, Y:integer) : any
mirrorZ		ni		(X:integer, Y:integer, Z:integer) : any

**Fig. 2.11** Method without Arguments

The row mirrorX does not contain a value in the column signature (Fig. 2.11). MirrorX flips the icon on the x-axis. It is a method that has no arguments and returns no value. You will call this method without parentheses.

Store. mirrorX;



**Fig. 2.12** Optional Parameter

The parentheses in the column signature indicate that `startPause` is a method. The data-type is given within square brackets. This means that the argument is optional, which results in two possibilities for the call:

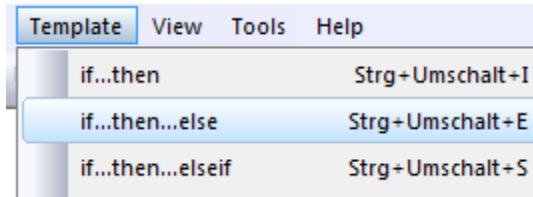
`Store.startPause;` -- no time limit

`Store.startPause(120);` -- pause for 120 seconds

Note: Some attributes are read-only; you can assign no value to them. Online help describes whether an attribute is read-only or not.

### 2.2.5 *Templates*

For a number of cases, Plant Simulation includes templates that you can insert into your source code or use as a starting point for developing controls. You reach Templates via the menu Template (Fig. 2.13):



**Fig. 2.13** Templates

First click through the method editor on the row in which the snippet should be inserted. Then click, e.g. Template—`if . . . then . . .` Under Select Template, you will find more templates. However, most templates in this selection will completely replace your source code. You can use the tab key to move through the template (between the areas in angle brackets).

### 2.2.6 *The Debugger*

The debugger helps you to correct your methods. Using the F11 key, you can quickly change between editor and debugger (or Run—Debug). Example: Open the method from the example above, place the cursor in the text and then press F11 (Fig. 2.14).

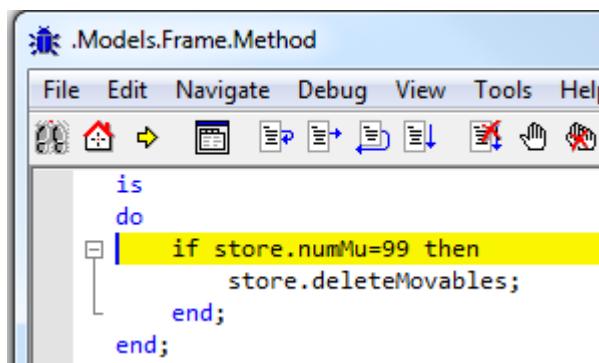


Fig. 2.14 Debugger

Using F11, you can execute your method in a stepwise manner. If the method is completed, it will open again in the editor. If an error occurs during the simulation, the debugger automatically opens and displays error messages. If you have made changes in the source code in the debugger, you can save the changes by pressing the F7 key.

## 2.3 SimTalk

SimTalk does not differentiate between uppercase and lowercase in names and commands. At the end of a statement, type a semicolon. Blocks are bounded by an "end" (no curly brackets as in Java or C++). If you forget the "end" Plant Simulation always looks for it at the end of the method.

### 2.3.1 Names

Plant Simulation identifies all objects and local variables using their name or path. You can freely select new names with the exception of a few key words. The following rules apply:

- The name must start with a letter. Letters, numbers or the underscore "\_" may follow
- Special characters are not allowed
- There is no distinction between capital and lowercase letters
- Names of key words and names of the built-in functions are not allowed

For method-blocks some names are reserved:

- **Reset:** Will be executed when clicking Reset in the EventController
- **Init:** Will be executed when clicking Init in the EventController
- **EndSim:** End of simulation (reaching the end time for simulation)

Generally, key words from SimTalk are prohibited names (all terms in SimTalk are highlighted in blue).

### 2.3.2 *Anonymous Identifiers*

SimTalk uses anonymous identifiers as wild cards. When running the method, these anonymous identifiers are replaced by real object references.

#### Root

The anonymous identifier "root" always addresses the top of the frame hierarchy. Starting from this frame, you can access underlying elements.

#### Example: Root

First, open the console in Plant Simulation: View—Toolbars and Docking Windows—Console (Fig. 2.15).

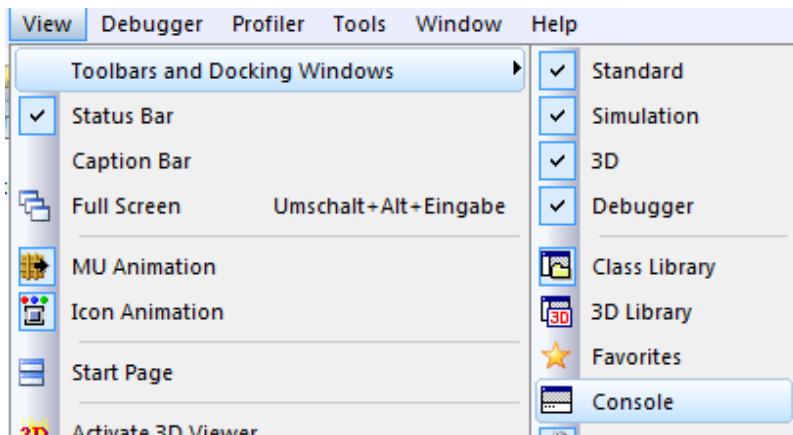


Fig. 2.15 Show Console

Create a new method. Use the method from the introduction example. Complete the method like this:

```
is
do
  -- writes the root name in the console
  print root.name;
end;
```

The console will show the name of the frame in which the method was placed.

#### Self

Self returns a reference to the current object (itself).

#### Example: Anonymous Identifier Self

```
is
do
  -- The name of the current method will be
  -- written to the console
  print self.name;
end;
```

## Current

Current is a reference to the current frame.

?

? denotes the object that has called the method (e.g. the object in which the method is entered as an exit control). The question mark allows a method to be used without modification by several objects.

@

@ refers to the MU that has triggered the method (you can access, e.g. all outgoing MUs).

## Basis

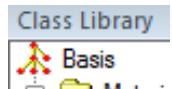
This denotes the class library. You can use it only in comparisons.

### 2.3.3 Paths

When objects are not located in the same frame or folder (name space), a path has to be placed in front of the name. Only the path allows for clearly identifying an object, so that it can be reached. A path is composed of names and periods (which serve as separators). Paths are divided into two kinds of paths:

- Absolute paths
- Relative paths

#### 2.3.3.1 Absolute Path



The starting point of the absolute path is the root of the class library. From here on objects are addressed from the "top" of the library. An absolute path always starts with a period.

Example: .Modelle.Frame.workplace\_1

#### 2.3.3.2 Relative Path

A relative path starts within the frame in which the method is located (without the first period).

Example: workplace\_1

Workingplace\_1 is located in the same frame as the method.

controls.Method1

This address refers to an object with the name "Method1" in a sub-frame "controls."

#### 2.3.3.3 Name Scope

All objects located in the same frame or folder form a name scope. Identical names are not allowed within a name scope. In other words, names in a name

scope may occur only once and all objects must have different names. In different frames, identical names may occur. Their path distinguishes the objects. If you try to assign a name twice, Plant Simulation shows an error message (Fig. 2.16).

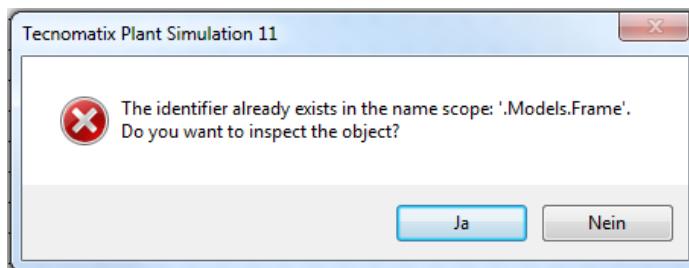


Fig. 2.16 Error message

### 2.3.4 *Comments*

Comments explain your source code. Plant Simulation distinguishes between two types of comments:

```
-- Comment until the end of line
/* Beginning of a comment, which
   extends over several lines
*/
```

Plant Simulation displays comments in green. It is recommended to comment on your source code. This comment should contain pertinent information about your method. Plant Simulation provides a template for this purpose: Template—Select Template. You can find the header comment in Code Snippets (Fig. 2.17).

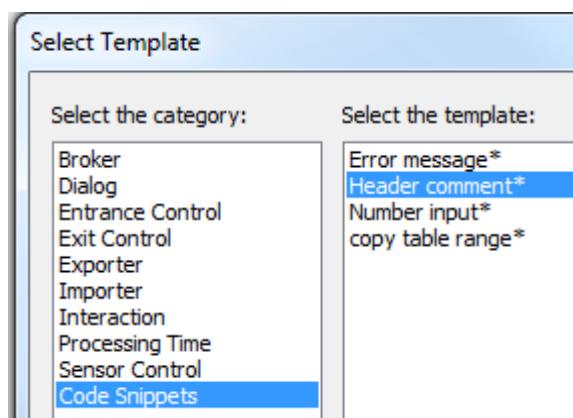


Fig. 2.17 Header comment

The header comment (started with “--” in front of the keyword is) is shown as a tooltip in the frame when you place the mouse on the method.

## 2.4 Variables and Data Types

If you want to store values in methods, you will need variables.

### 2.4.1 *Variables*

A variable is a named location in memory (a place where the program stores information). Variables should have meaningful symbolic names. You must first declare a variable (introduce its name), before you can use it. Plant Simulation distinguishes between local and global variables. A local variable can only be accessed within the respective method. This variable is unreachable for other methods. Global variables are visible for all methods (in every frame). You can set values and get values from all methods. In this way, you can organize data exchange among the components.

### 2.4.2 *Local Variables*

Local variables are declared between "is" and "do." A declaration consists of the name of the local variable, a colon and a data type. The keyword "do" follows the last declaration.

Example:

```
is
  Name : Type;
do
-- statements
end;
```

For instance:

```
is
  stock_in_store : integer;
do
..-statements
end;
```

The declaration reserves an address in the memory and you define access on it by a name. This is why the operating system must know what you want to save. A true/false value requires less memory (one bit) than a floating-point number with double precision (minimum 32 bit). Information about the memory size is revealed through the so-called data types. The data type determines the maximum value range of variables and regulates the permissible operations. A value is assigned to a variable using :=.

### Example: Declaring Variables

The circumference of a circle is to be calculated from a radius and Pi. Pi is defined in Plant Simulation and can be retrieved via Pi. The result is written on the console using the print command.

```
is
  radius :integer; --integer number
  circum :real; --floating point number
do
  radius:=200;
  circum:= radius*PI;
  print circum;
end;
```

SimTalk provides the following data types: acceleration, any, boolean, date, datetime, integer, length, list, money, object, queue, real, speed, stack, string, table, time, timeSequence, and weight (Table 2.2).

**Table 2.2** Data Types

Name	Range of values
acceleration	real, m/s <sup>2</sup>
any	the data type will be determined only after the assignment of the value (like VB: variant)
boolean	TRUE or FALSE
integer	-2,147,483,648 to 2,147,483,647
real	floating-point numbers
string	character (each letter, numbers)
object	reference to an object (except comment)
table	local variable with the behavior of a table
list	see above
stack	see above
queue	see above
money	as real, the value is interpreted as currency
length	as real, the value is interpreted as meters
weight	see above, kg
speed	real → m/s
time	real → seconds.; output: <hh>:<mm>:<ss.sss>
date, datetime	date from 1.1.1970 to 31.12.2038

Data types can be converted to a limited extent. Plant Simulation initializes all local variables automatically. The value depends on the data type (Table 2.3).

**Table 2.3** Initializing Data Types

Type	Initialization
boolean	FALSE
integer	0
real	0
string	"" (empty string)
object	void
table	void
list	void
stack	void
queue	void
money	0
length	0.0
weight	0.0
speed	0.0
time	0:00:00
date	1.1.1970 0:00:00

If you want to define another start value in the simulation, you can use the `init` method.

### 2.4.3 Arrays

With the help of arrays, you can access values in a one- or two-dimensional value field using one name and indices. The values of an array have the same data type. Arrays can be declared as follows:

Name: `<data type>[];`

Within square brackets, you can determine the size of the array. If you do not define a quantity, then the size of the array is initially indeterminate. The smallest index within an array has an index of one. Example:

```
is
  arr:integer[3]; --an array with 3 integer-values
do
  -- set values
  arr[1]:=10;
  arr[2]:=11;
  arr[3]:=12;
  -- output the second value
  print arr[2];
end;
```

You can use the method `makeArray (<value1>, <value2>, ...)` to convert a set of values in an array. Example:

```
is
  arr:integer[]; --an array of integer-values
do
  -- convert values into an array
  arr:=makeArray(13,14,15);
  -- print the second value
  print arr[2];
end;
```

You can also define two-dimensional arrays. The declaration of a two-dimensional array contains x and y dimensions in square brackets. When setting and reading the data, you have to enter two indices accordingly. Example:

```
is
  arr:integer[2,2]; --4 values
do
  -- set values
  arr[1,1]:=1;
  arr[2,1]:=2;
  arr[1,2]:=3;
  arr[2,2]:=4;
  -- output the value on the position 2,2
  print arr[2,2];
end;
```

#### 2.4.4 *Global Variables*

The local variables are accessible only from the method in which they are declared. For clarification, the following example is provided.

You need two methods in a frame (Method1 and Method2, Fig. 2.18).



**Fig. 2.18** Example Frame

In Method2, define a variable named "number" of the type integer. Assign the value 11 to the variable. Method2:

```
is
  number:integer;
do
  number:=11;
end;
```

In Method1, try to read the variable "number" and write the value of "number" to the console. Method1:

```
is
do
    print number;
end;
```

If you run Method1 (F5 or Run—Run), you get an error. It opens the debugger and the faulty call is highlighted. The error description is displayed in the status bar of the debugger (Fig. 2.19).

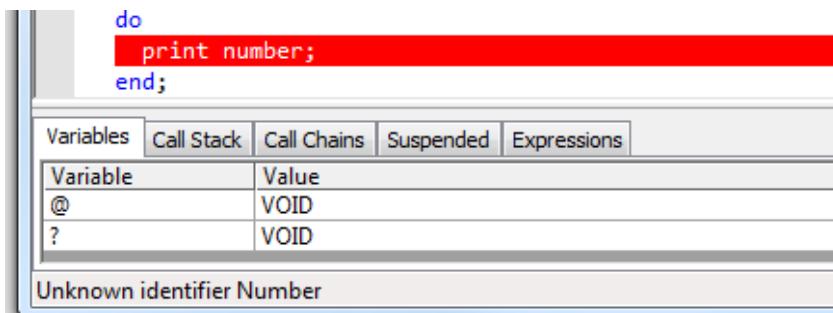


Fig. 2.19 Debugger Error Message

Method1 cannot access a variable "number" of Method2. If you need the data in several methods, you have to define a global variable (variable object from the folder information flow) to exchange data. All methods can set and access the value of these variables. The global variable is defined as an object in the class library and is addressed just like the other objects by name and path. You have to determine the data type of the variable and can specify a start value.

#### Example: Global Variable

Insert a global variable into the frame above. Rename the variable as "number," data type integer (start value remains zero, Fig. 2.20).

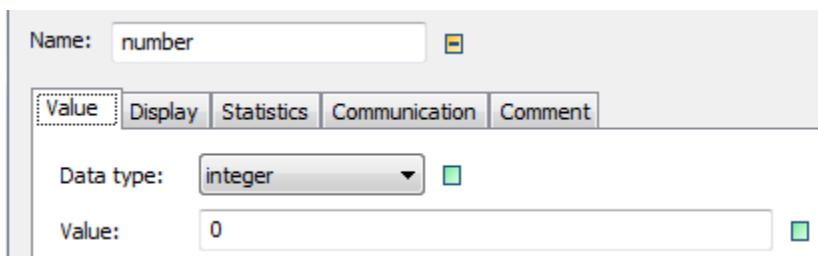


Fig. 2.20 Global Variable

Change Method2 deleting the variable declaration of "number"; leave the rest unchanged:

```
is
do
  number:=11;
end;
```

Now, start Method2 (the value of the global variable would have to change), and then Method1. The value of the global variable is displayed in the console. Global variables are reset to the start value when you press the Reset button if you select the option Initial Value and specify a start value. In the previous example, the value will be set to zero when you enter the setting from Fig. 2.21.



Fig. 2.21 Initial Value

## 2.5 Operators

Using a combination of constants and variables, you can define complex expressions (e.g. calculations). Operators are used for concatenating expressions. Plant Simulation differentiates between:

- Mathematical operators
- Logical (relational) operators
- Assignment operators

Logical operators are needed for comparisons.

### 2.5.1 Mathematical Operators

SimTalk recognizes the following mathematical operators:

- Algebraic sign, subtraction
- \*
- / Division
- // Integer division
- \ Modulo (remainder of integer division)
- +
- Addition(concatenation of strings)

The integer division (which is only defined for the data type integer) always delivers a whole number. Any decimal places are suppressed. When calculating data for the data type real, the result is an output of up to seven valid digits (eighth digit rounded, working with decimal power).

### 2.5.2 *Logical (Relational) Operators*

**Table 2.4** Logical Operators

Operator	Function	Result
AND	logical AND	TRUE, if all expressions are TRUE
OR	logical OR	TRUE, if at least one expression is TRUE
NOT	Not	Invert the Boolean value
<	Less than	
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
=	equal	
!=	unequal	

A logical expression is always interpreted from left to right. The evaluation is completed once the value of an expression is established.

#### Example: Logical Operators

Create a simple method (+ show the console).

```
is
  num1:integer;
  num 2:integer;
  num 3:integer;
  val1:boolean;
  val2:boolean;
  val3:boolean;
do
  num1:=10;
  num2:=23;
  num3:=num1*num2;
  val1:=num1<num2;
  val2:=num1<num2;
  val3:=val1 AND val2;
  print val3;
end;
```

Examine the operators from Table 2.4. Let the console show different variables (Save + F5).

### 2.5.3 *Assignments*

The operator ": =" assigns a new value to a variable. First, the expression to the right of the operator is calculated. If the value and the variable have the same data type, then the value is assigned to the variable.

### Example: Variable—Value Assignment

```
is
  num:integer;
do
  num:=1;
  num:=num+1;
  -- first num+1, then assignment to num
  print num;      -- print the new value of num
end;
```

If the data types are different, then the values have to be converted. Plant Simulation automatically converts real into integer and vice versa. For other types of data, you need to use type conversion functions.

### Example: Type Conversion 1

```
is
  num:integer;
  text:string;
  res:integer;
do
  num:=10;
  text:="20";
  res:=num{text;
  print res;
end;
```

You receive an error message when you run the method (Fig. 2.22).

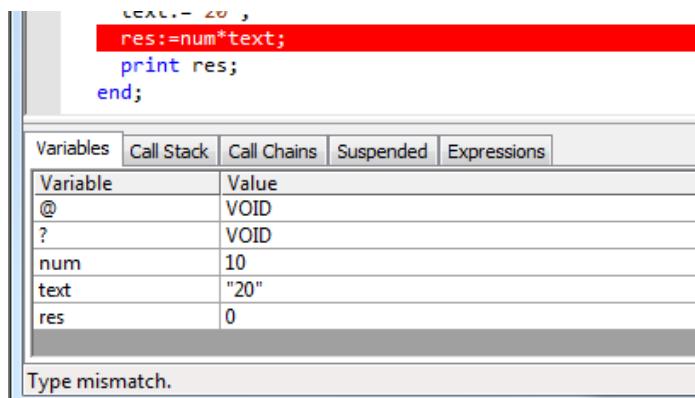


Fig. 2.22 Debugger Message

You need to convert the data into the right data type. The most important type conversion functions are listed in Table 2.5.

**Table 2.5** Data Type Conversion Functions

Syntax	Return value of data type
bool_to_num(<boolean>)	real
num_to_bool(<integer>)	Boolean
str_to_bool(<string>)	Boolean
str_to_date(<string>)	time
str_to_datetime(<string>)	datetime
str_to_length(<string>)	length
str_to_num(<string>)	real
str_to_obj(<string>)	object
str_to_speed(<string>)	speed
str_to_time(<string>)	time
str_to_weight(<string>)	weight
to_str(<any>, ...)	string

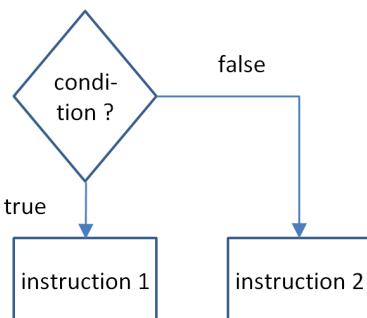
In the example above, a conversion from string to integer is required. You can realize a type conversion of text with the help of the function `str_to_num` (...). The method has the following syntax:

`str_to_num (text)`

Extend the example:

```
is
  num:integer;
  text:string;
  res:integer;
do
  num:=10;
  text:="20";
  res:=num*str_to_num(text);
  print res;
end;
```

## 2.6 Branching



After testing a condition, the branch decides which of the following instructions should be executed (Fig. 2.23). If the condition is met, the if branch (true) will be executed. If the condition is not met, the else branch (false) will be executed.

**Fig. 2.23** Branching

The general syntax is as follows:

```
if condition then
    instruction1;
else
    instruction2;
end;
```

**Example: Branch 1**

You only need one method for the example. We want to query if value1 is less than 10. If yes, then a message "If branch is executing" should be displayed in the console; otherwise, "Else branch is executing" should appear.

```
is
    value1:integer;
do
    value1:=12;
    if value1< 10 then
        print " If branch is executing";
    else
        print " Else branch is executing";
    end;
    print " Here we continue normally";
end;
```

Try different queries.

After passing through the branch, the execution of the code continues. If more than one condition is to be checked, the conditions can be nested. The nesting depth is not limited. In this case, a new condition begins after the if branch with "elseif".

**Example: Branch 2**

Extend the example above:

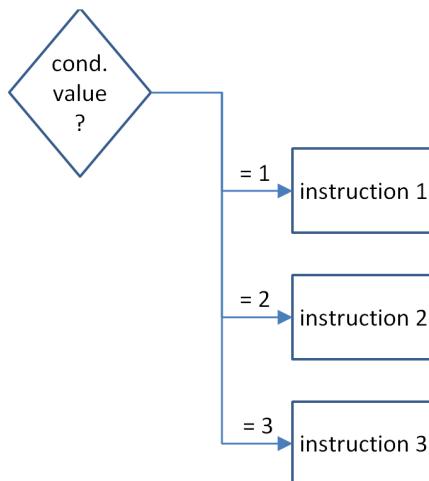
```
is
    value1:integer;
do
    value1:=7;
    if value1= 10 then
        print " value1 is 10.";
    elseif value1=9 then
        print " value1 is 9";
    elseif value1=8 then
        print " value1 is 8";
    elseif value1=7 then
        print " value1 is 7";
    -- and so on.....
    end;
    print "Here we continue normally. ";
end;
```

If you have to check many conditions, this construction quickly becomes complicated. Then, you can use the so-called “case differentiations.”

## 2.7 Case Differentiation

The case differentiation checks, for example, the value of a variable and assigns actions to certain values (Fig. 2.24). Case differentiation in SimTalk has the following syntax:

```
inspect <expression>
  WHEN <constant_1> THEN <instruction 1>
  WHEN <constant 2> THEN <instruction 2>
  -- ...
end;
```



**Fig. 2.24** Case Differentiation

### Example: Case Differentiation

```
is
  num:integer;
do
  num:=2;
  inspect num
    when 1 then print "Num is 1.";
    when 2 then print "Num is 2.";
    when 3 then print "Num is 3.";
    -- and so on
    else
      print "Not 1, not 2, not 3 !";
  end;
end;
```

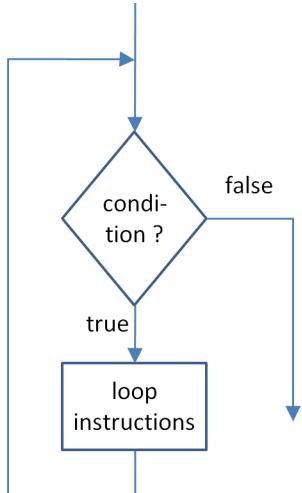
## 2.8 Loops

To execute a statement repeatedly, use a loop command.

### 2.8.1 Conditional Loops

Conditional loops are executed as long as a condition is met or not met. Plant Simulation provides various conditional loops.

#### 2.8.1.1 Header-Controlled Loops



Before passing through the loop instructions, Plant Simulation checks whether a condition is met or not. The loop is repeated if the validation of the condition returns as true. If the condition before the first loop is not met, the loop instructions will not be executed. Make sure that the loop condition is false some of the time (e.g. increase in the value of a variable, until the value exceeds a certain limit).

Endless loops are terminated with the key combination **Ctrl + Alt + Shift**.

**Fig. 2.25 Loop**

Syntax:

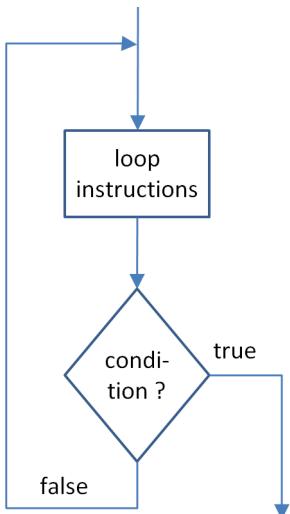
```
while <condition> loop
  <instructions>
end;
```

#### Example: While Loop

Loop 1, Loop 2 to Loop 10 should be written to the console.

```
is
  i:integer;
do
  i:=1;
  while i<10      loop
    print "Loop run number:" + to_str(i);
    i:=i+1;
  end;
end;
```

### 2.8.1.2 Footer-Controlled Loops



The condition is checked after the execution of the loop statement. If the condition for a termination of the loop is not met, the loop statement will be executed one more time (Fig. 2.26). The loop statement is executed at least once.

Fig. 2.26 Loop

Syntax:

```
repeat
  -- instructions
until <break condition is met>;
```

#### Example: Repeat Loop

```
is
  i:integer;
do
  i:=1;
  repeat
    print "Loop number:" + to_str(i);
    i:=i+1;
  until i>5;
end;
```

### 2.8.2 For Loop

If you know exactly how often the loop is to be iterated, use the for loop or from loop. These need a running variable to control the number of runs of the loop. The variable is increased or decreased during each run starting from an initial value by a certain value. When a predetermined end value is reached, the loop will be terminated.

Syntax1 :

```
from < Initialization > until <condition> loop
    <instructions>
end;
```

Syntax 2:

```
for < Initialization > to <end value> loop
    -- loop instructions
next;
```

### **Example: From Loop**

Outputs will be shown in the console (loop 1, loop 2 and so on)

```
is
    i:integer;
do
    from i:=1; until i=10 loop
        print "loop " + to_str(i);
        i:=i+1;
    end;
end;
```

### **Example: For Loop**

A loop should be executed five times:

```
is
    i:integer;
do
    for i:=1 to 5 loop
        print "loop " + to_str(i);
    next;
end;
```

You can count in the loop "backwards with "downto" instead of "to".

### **Example: For- Loop with downto**

This is similar to the previous example:

```
is
    i:integer;
do
    for i:=5 downto 1 loop
        print "loop " + to_str(i);
    next;
end;
```

With the `ExitLoop <integer numberOfLoops>` command, you can stop the loop at runtime.

## 2.9 Methods and Functions

A function is the definition of a sequence of statements, which are processed when the function is called. There are functions in different variants: Arguments are passed to some functions, but not to others. Arguments are values that must be passed to the function, so the function can meet its purpose. Some functions give back a value, while others do not.

### 2.9.1 Passing Arguments

Arguments serve the purpose of passing data on during the function call (not just a stock removal function call, for example, but also the number of the average stock removal per day is handed over by function call). The data type for the given value must match the data type of the argument declared in the function. The arguments have to be declared in the function. The declaration will be made at the beginning of the method before "is" in parenthesis in the following format:

```
(name : type)
```

For example:

```
(Stock_removal : integer)
```

Within the method, the arguments can be used like local variables, with the caller determining the initial values.

#### Example: Passing Arguments 1

The user is to enter the radius of a circle and the size of the circumference is to be displayed in the console. To enter the argument, you need a text box. This is called by the function "prompt." Using the method "prompt," you can ask the user for input. If you pass a string to the method "prompt," then this string will be shown as a command prompt (Fig. 2.27).

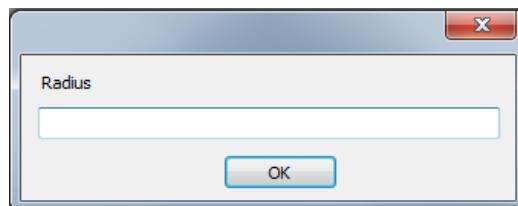


Fig. 2.27 Prompt

Since the type of the data to be read is string, a type conversion is necessary (str\_to\_num (identifier)). Method test:

```

is
  radius :string;
  circumference : real;
  val:real;
do
  -- prompt
  radius:=prompt ("Radius");
  -- type conversion to real
  val:=str_to_num(radius);
  -- calculate circumference and display in
  --the console
  circumference:=val*2*PI;
  print  circumference;
end;

```

Open the console to see the result.

### 2.9.2 *Passing Several Arguments at the Same Time*

Within the definition, a semicolon separates several arguments. When you call the function, you have to pass the same number of arguments that you have defined in the function.

#### **Example: Passing Arguments 2**

For two given numbers, the larger number is to be returned after the call of the function. The function should be named getMax (number1, number2).

```

(number1:integer;number2:integer)
-- passing arguments
:integer --type of the return value
is
do
  if number1 >= number2 then
    result:=number1; -- return value
  else
    result:=number2;
  end;
end;

```

Call the function (from another method):

```

is
do
  print getMax(85,23);
end;

```

### 2.9.3 *Result of a Function*

Methods can return results (a method that returns a value is called a function). For this purpose, you have to enter a colon and the type of the return value before "is." The result of the function has to be assigned within the function to the automatically declared local variable "result." Another possibility is to use "return." Return passes program control back to the caller. You can also pass a value on this occasion. After processing, the function will return the content of the variable "return" to the caller (the return value will replace the function call).

#### **Example: Results of a Function**

A function "circumference" is to be written. The radius will be passed to the function and the function returns the circumference.

Function circumference:

```
(radius:real) -- argument radius (2)
:real -- data type of the return value
is
do
    result:=radius*2*PI; -- (3)
end;
```

Method test (in the same frame):

```
is
    res:real;
do
    res:= circumference (125); -- (1)
    print res;
end;
```

Explanation:

- (1) A value is given when calling the function.
- (2) The function declares the arguments.
- (3) The function inserted the value passed at the designated position.

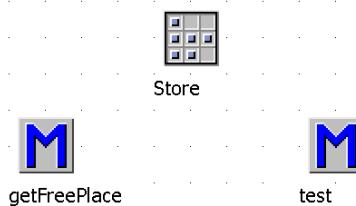
If you want to return more than one result from one function, the result will not work. One solution is to define arguments as a reference. Usually, the program makes copies of the data and the function continues to work with the copies (except if you are passing objects; objects will always be passed as reference to the object). The original values of simple data types remain unchanged in the calling method. If arguments are defined as reference, you can change the values in the calling method by the called function. The definition is accomplished with:

```
(byref name:datatype)
```

#### **Example: Argument Passing byRef**

You should write a method that returns the first free place in a store. The storage place is addressed through an x- and a y-coordinate. The method to access

the individual place of the store is `<store>.pe(x, y)`. You can check using the method `Empty` if the place is still free. Create a simple frame according to Fig. 2.28.



**Fig. 2.28** Example Frame

The store has an x-dimension of 10 and a y-dimension of 10 (100 places). The method `getFreePlace` might look like this:

```
(byref x:integer;byref y:integer)
is
  i:integer;
  k:integer;
do
  -- look for all places
  for i:= 1 to store.xDim loop
    for k:=1 to store.yDim loop
      if store.pe(i,k).leer then
        --if place is free --> store in the variables
        x:=i;
        y:=k;
        -- terminate method
        return;
      end;
    next;
  next;
end;
```

In the method, the function result is passed to the transfer parameters `x` and `y`. In the calling method, two parameters must be defined as local variables and are passed at the method call. The method `test` could appear as follows:

```
is
  --define local variables
  x:integer;
  y:integer;
do
  --call method, pass local variables
  getFreePlace(x,y);
  -- check
  -- the local variables are changed by the method
```

```

print x;
print y;
end;

```

Call the method test. Drag entities from the class library to the store. Then call the method test again. The method getFreePlace should find in each case the next free place.

### 2.9.4 Predefined SimTalk Functions

SimTalk has a range of ready-to-use functions (Table 2.6, Table 2.7).

**Table 2.6** Functions for Manipulating Strings

Function	Description
copy(<string>,<integer1>,<integer2>);	The function "copy" copies a number of characters (integer2) from a string starting from the position integer1. The first character has the position 0.
incl(<string1>,<string2>,<integer>);	The function "incl" inserts a string2 into string1 before the position integer. The new string is returned.
omit(<string>,<integer1>,<integer2>);	"omit" copies the string and deletes the substring within it from starting position integer1 with (integer2) characters. The new string will be returned.
pos(<string1>,<string2>)	"pos" shows the position within string2, in which string1 occurs for the first time. If string1 is not contained in string2, zero is returned; otherwise, the position as integer.
strlen(<string>)	"strlen" returns the length of the string passed.
trim(<string>)	This removes leading and trailing blanks and tabs from the given string, and returns the modified string.
splitString(<string>,<separatorString>)	"splitString" splits a string on the separatorString and returns an array of strings (new in Version 11 TR3).

#### Example: Functions for Manipulating Strings

The file extension of a filename is to be identified and re-turned. The dot will be searched for first. Then, starting from a position after the dot all characters until the end of the string will be copied. The result will be displayed in the console.

```

is
  filename, extension:string;
  length, posPoint:integer;
do
  filename:="samplefile.spp";
  -- search point

```

```

posPoint:=pos(".",filename);
-- find out the length of the text
length:=strlen(filename);
-- copy the substring
extension:=copy(filename,posPoint+1,
    length-posPoint);
-- display in the console
print extension;
end;

```

### Example: Method Split

The method to be developed should take a string and a delimiter as arguments and return a list of individual data. To create the split function, you need the methods Pos (for searching) and Copy. The split method has to search the positions of the delimiter in the given string and copy the substring between the delimiters into a list. The list with the substrings or void is returned. Therefore, the function might appear as follows:

```

(text:string;delimiter:string):list
is
    list_:list[string];
    posI:integer;
do
    list_.create;
    --check the position of the delimiter
    posI:=1;
    while(posI>0) loop
        posI:=pos(delimiter,text);
        if posI=0 then
            --delimiter doesnt exists
            --append the complete text in the list
            list_.append(text);
        else
            --copy substring
            list_.append(copy(text,1,posI-1));
            -- shorten the text
            text:=omit(text,1,posI);
        end;
    end;
    return list_;
end;

```

A test of the split method could look like this:

```

is
do
    local strings:=split("1,2,3,4",",");
    print strings.read(2);
end;

```

**Table 2.7** Mathematical Functions

Function	Description
abs(x)	Absolute amount of x
round(x)	Rounds x to the nearest whole number (on or off)
round(x,y)	Rounds x on y digits
floor(x)	Nearest whole number less than or equal to x
ceil(x)	Nearest whole number greater than or equal to x
min(x,y)	Minimum
max(x,y)	Maximum
pow(x,y)	$x^y$
...	

### 2.9.5 Method Call

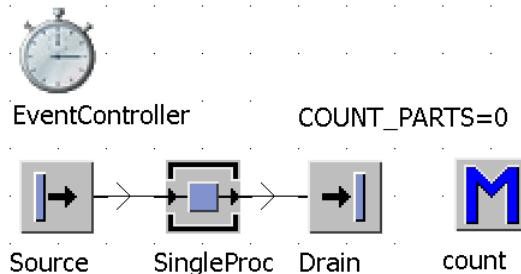
Methods are called by their names. During the simulation, methods have to be called often in connection with certain events. For this purpose, Plant Simulation provides several ways to call methods.

#### 2.9.5.1 Sensors

Most objects are equipped with sensors that trigger methods when an MU enters the object or exits. Length-oriented objects, such as the line or the track, have separate sensors for moving forward or in reverse.

##### Example: Method Calls by Sensors

Parts manufactured by a machine are to be counted using a global variable. Create a simple frame as in Fig. 2.29.

**Fig. 2.29** Example Frame

"COUNT\_PARTS" is a global variable of type integer and an initial value of zero. The method "count" should look like this:

```
-----| increases the global variable "COUNT_PARTS" with
-----| each call by one
-----|
is
do
  COUNT_PARTS := COUNT_PARTS +1;
end;
```

The method should now be called if a part has exited the SingleProc. Open the dialog of the SingleProc and select the tab Controls (Fig. 2.30).

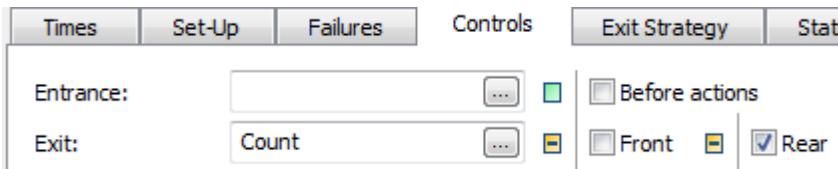


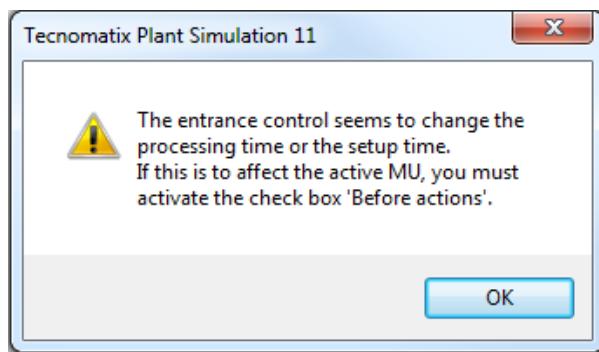
Fig. 2.30 Exit Control

Select the method Count in the field "Exit" and activate the rear option.

The main sensors are entrance and exit control. The setup control is triggered when a setup process starts and ends. You can activate the sensor with the front or the rear. The choice depends on the actual case. The rear exit control is triggered when the part has already left the object. For counting purposes, this is the right choice. If you trigger the control with the front, the MU is still on the object. If the subsequent object is faulty or still occupied, the front sensor is triggered twice (once when trying to transfer, the second time when the transfer is complete). If you select "front" in the exit control, then you need to trigger the transfer using SimTalk, even if a connector leads to the next object (e.g. `@.move` or `@.move(successor)`). If you press the F2 key in a field in which a method is registered, Plant Simulation will open the method editor and load the relevant method. If you press the F4 key, Plant Simulation creates an internal method and enters this method as control.

### Before Actions

Some actions must be performed before the MU is finally moved to the block. This applies particularly, among others, to changes in the processing or setup time or in assembly lists. The EventController calculates, e.g. at the entrance of the MU, its exit time. If you make the changes only after the entrance of the MU, then this is not considered in the EventController for the current process on the block. Starting with Version 10, you get a notice from Plant Simulation in such cases (Fig. 2.31).

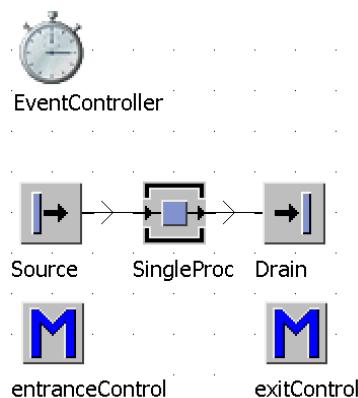


**Fig. 2.31** Notice before Actions

With the option "Before actions," the control is executed before the MU is transferred to the block.

#### Example: Entrance Control before Actions

The following example illustrates the function. Create a frame like in Fig. 2.32.



**Fig. 2.32** Example Frame

Let the source remain unchanged. The method entranceControl is the entrance control and the method exitControl is the exit control of the SingleProc. The methods have the following content (Method entranceControl):

```
is
do
  ?.procTime:=10;
end;
```

Method exitControl:

```
is
do
    ?.procTime:=1;
    @.move;
end;
```

Set in the EventController an end of one hour. Let the simulation run without the option "Before actions." After the end of the simulation, examine the statistics of the drain. You will see that the change in the processing time of the entrance control had no effect (nearly 3,600 parts per hour—i.e. one second of processing time). Now activate the option "Before actions" (Fig. 2.33).

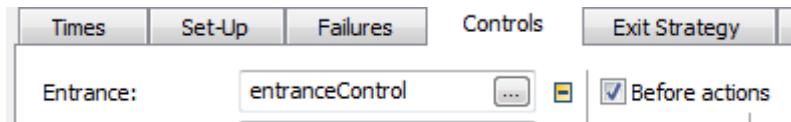


Fig. 2.33 Option Before Actions

If the simulation now runs for one hour, about 360 parts are manufactured (10 seconds of processing time). The change in the processing time is now made before the movement of the MU and is effective for the respective MU.

### 2.9.5.2 Other Events for Calling Methods

For calling methods, you can use other events. These events can be found in the object dialogs under the command Tools—Select Controls (Fig. 2.34).



Fig. 2.34 Tools—Select Controls

The failure control, for example, is called if a failure of the object begins and ends (when the value of the attribute failure changes).

#### Example: Fail Control

You are to simulate the following part of a production. On a multilane, non-accumulating line, parts are transferred and transported three meters. Processing on each lane by a separate machine follows. If one machine fails, the entire line must be stopped (all lanes have to be stopped). Create a frame according to Fig. 2.35.

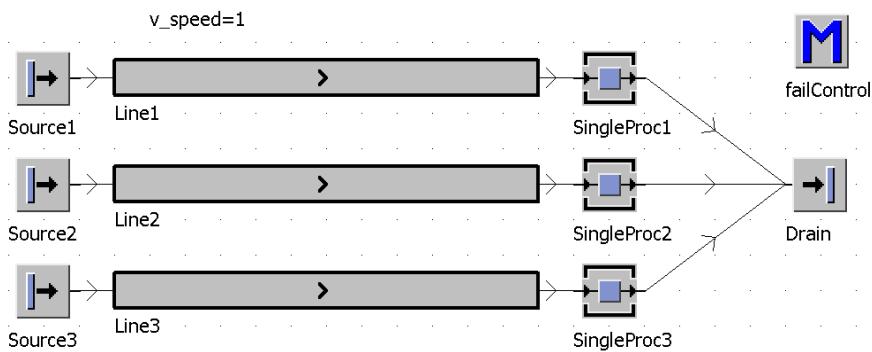


Fig. 2.35 Example Frame

Settings: Source1 to Source3: interval of one second, no blocking; conveyors: 12 m, 1 m/s; SingleProc: one second of processing time each; failures: 90 per cent availability, 10 minutes MTTR; drain: zero seconds of processing time.

One way to stop all lanes by one machine failure could be when the method "failControl" checks whether a machine has failed. If at least one machine fails, the speed of all three lines is set to zero. If no machine fails, the speed is set to the value that is stored in the global variable "v\_speed."

Method failcontrol:

```

is
do
  if SingleProc1.failed or
    SingleProc2.failed or
    SingleProc3.failed then
    Line1.speed:=0;
    Line2.speed:=0;
    Line3.speed:=0;
  else
    Line1.speed:=v_speed;
    Line2.speed:=v_speed;
    Line3.speed:=v_speed;
  end;
end;

```

Now the method "failcontrol" must be called whenever a failure occurs, or when the failure ends. Click in the dialog of the SingleProc objects Tools—Select Controls: Enter the method "failcontrol" as the failure control (Fig. 2.36).



Fig. 2.36 Fail Control

Repeat this for all three machines. The result can very well be proven statistically. In theory, the individual machines can work 90 per cent of the time. If the failures of the other machines result in the effect that no more parts are being transported by the conveyors, then the machines remain without parts for 20 per cent of their processing time, in addition to their own failures. This means that the machines can, on average, operate at only 70 per cent. Run the simulation for a while (for at least two days), and then click the tab Statistics in SingleProc1, 2 or 3.

### 2.9.5.3 Constructors and Destructors

A constructor control is called when you insert a particular object into a frame. The destructor is called when you delete an object in a frame. The destructor is called when you delete an object in a frame. You can use constructors to initialize or create object tables for later, easy access. Constructors and destructors must be programmed in the class library. The constructor and destructor controls are assigned via Tools—Select Controls.

#### Example: Constructor and Destructor

If a SingleProc is inserted into a frame, an object reference should be inserted automatically into a table. When the machine is deleted, it should also be deleted from the table. Insert a TableFile (objectList) into an empty frame. Set the first column to the data type object. Then duplicate a SingleProc in the folder "Models." Add to the duplicate two user-defined attributes (constructor, destructor, both data type methods). Select in the duplicated SingleProc, Tools—Select Controls and choose the concerned methods for destructor and constructor (Fig. 2.37).

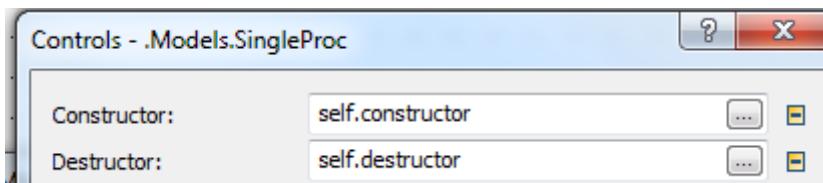


Fig. 2.37 Constructor and Destructor

Method constructor:

```
is
do
  -- insert object into the objectList
  --access to the object: ?
  root.objectList.writeRow(1,
    root.objectList.yDim+1, ?);
end;
```

Method destructor:

```
is
do
  -- look for the object in the table
  root.objectlist.setCursor(1,1);
  if root.objectlist.find(?) then
    -- if found - delete row
    root.objectList.cutRow(root.objectlist.cursorY);
  end;
end;
```

#### 2.9.5.4 Drag and Drop Control

The drag and drop control you must generally define for the object on which you want to "drop" the elements. The drag and drop control must have a transfer parameter of type String. Plant Simulation passes as an object the path of the (dropped) object. You can use the method str\_to\_obj to transfer the string into an object reference.

##### Example: Drag and Drop Control

When you drop an object on a table, this object should be entered in this table. Add a table in an empty frame. Turn off the inheritance and format the first column as object. Select Tools—Select Controls—Drag and Drop. Press the F4 key. Plant Simulation generates an internal method (Fig. 2.38). Accept your changes.



Fig. 2.38 Drag and Drop Control

Method onDragDrop:

```
(dropObj:string)
is
do
  self.~.writeRow(1,self.~.yDim+1,
    str_to_obj(dropObj));
end;
```

#### 2.9.5.5 Method Call after a Certain Timeout

You can call SimTalk methods after a certain timeout. This can be useful if you need to trigger calls after an interval refers to an event. The EventController generates an event of type MethCall and integrates it into the event list. You will find these entries in the event debugger of the EventController (Fig. 2.39).

Event Debugger - 0.0000				
Breakpoints active		Breakpoints	Trace File:	
B	Type	Time	Receiver	Sender
	CreateMU	0.0000	Source	
	Out	0.0000	.MUs.Entity:3	Drain
	MethCall	10:00.0000	Method	init

Fig. 2.39 Event Debugger

### Example: Ref-methCall

In this simulation, the machine should send a status message (ready) 10 seconds before the completion of a part (e.g. to inform a loading device). The message should be shown first as an output in the console. Create a simple frame as in Fig. 2.40.

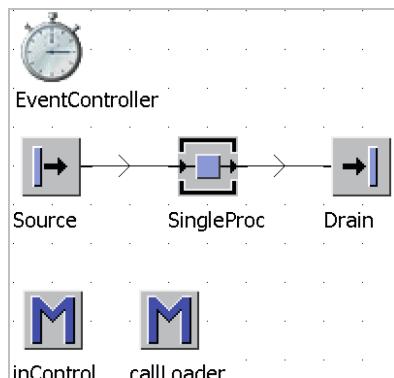


Fig. 2.40 Example Frame

Settings: Source interval two minutes, SingleProc two minutes processing time, inControl as entrance control SingleProc. The method callLoader writes a short message into the console. Method callLoader:

```
is
do
  print time_to_str(eventController.simTime) +
    " called loader";
end;
```

The call of the method callLoader should now take place 10 seconds before the completion of the part on the SingleProc. If you use the entrance control to start the timer, the method should be started after the processing time of SingleProc minus 10 seconds. For calling a method after a certain period, use in Plant Simulation the method <ref>.MethCall (<time>, [<parameter>]).

You cannot address the method with a path or directly with a name, because the method is then called directly. Instead, use the method ref (<path>). This returns a reference to the method by which you gain access to it. In the example above, the method InControl should look like this:

```
is
do
  ref(callLoader).methcall(SingleProc.procTime-10);
end;
```

### 2.9.5.6 Recursive Programming

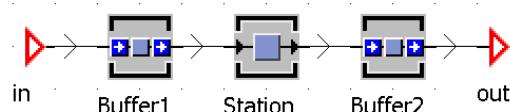
Methods are called recursive if they call themselves to perform certain work. Normally, parameters are passed for the recursion. Recursive methods can be used to program mechanisms, for example, to browse nested network structures. Analogous to endless loops, recursions carry a risk of "endless" method calls. In contrast to endless loops, "endless recursions" cannot be quit using a keyboard shortcut. For the next example, the general methods from Table 2.8 are required:

**Table 2.8** Common Methods and Attributes of Plant Simulation Objects

Method/ Attribute	Description
<path>.class	Class of the object
<path>.class.name	Name of the class of the object
<path>.internalClassName	Internal class name of the object—with this method you can easily read the basic type of the object (e.g. network); the internal class names can be found most easily via the attribute window of the objects
<path>.resourceType	Resource type of the object; possible values are Production, Transport or Storage
<path>.numNodes	Returns the number of objects of a frame
<path>.node(<integer>)	Returns the object with the index <integer>. The order of insertion into the frame determines the index

### Example: Recursive Programming

In a frame, all SingleProcs and ParallelProcs are to be read. Their locations and the current processing times should be written into a table. Each production consists of networks (workstation), each workstation of two buffers and a single station. Create the workstation (as frame) according to Fig. 2.41.



**Fig. 2.41** Sub-frame Workstation

Create using the workstation a frame as in Fig. 2.42.

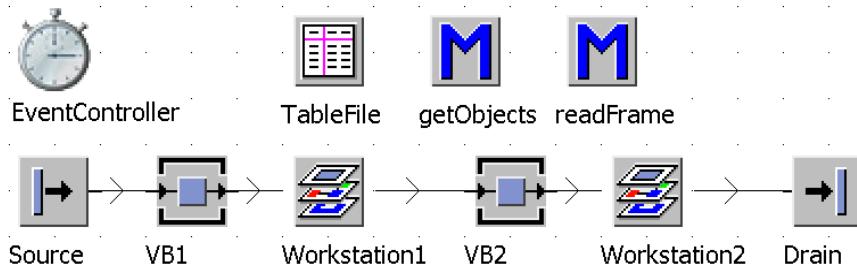


Fig. 2.42 Example Frame

Format the TableFile like in Fig. 2.43.

	string	time
1	1	2
string	path	processing time
1		

Fig. 2.43 Format TableFile

The method `getObjects` calls the method `readFrame` and passes a reference to the root frame. The method `readFrame` checks all objects of the frame. If the `internalClassName` of the object is either "Place" or "Machine," then the path and the processing time will be inserted into the table. If the `internalClassName` is "Network," the method `readFrame` is called again and passed as a reference to the appropriate frame. The calling method waits for the end of the called method. In this way, the entire frame is searched. Method `getObjects`:

```
is
do
  tableFile.delete;
  readFrame(root);
end;
```

Method `readFrame`:

```
(frame:object)
is
  i:integer;
do
  for i:=1 to frame.numNodes loop
    if frame.node(i).internalClassName="Place" or
    frame.node(i).internalClassName="Machine" then
      root.tableFile.writeRow(1,
        root.tableFile.yDim+1,
        to_str(frame.node(i)),
        frame.node(i).procTime);
```

```

elseif frame.node(i).internalClassName="Network"
  then
    -- recursion
    root.readFrame(frame.node(i));
  end;
next;
end;

```

### 2.9.5.7 Observer

Plant Simulation can watch observable values on its own and invokes a method when the value is changing. The name of the property (as a string) and value (before the change) will be passed. You can access the object using “?”.

#### Example: Observer

You should count the parts on a conveyor line. A variable is used to display the number of MUs. Use an observer for this purpose. Create a frame as in Fig. 2.44.

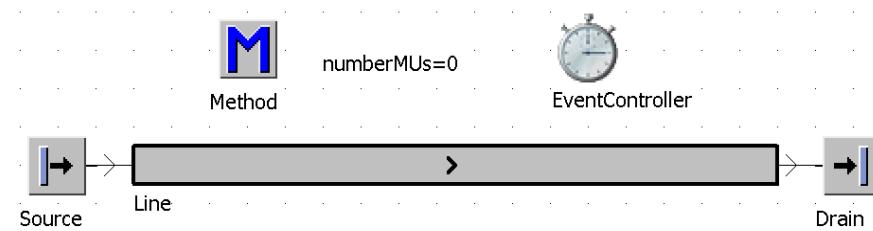


Fig. 2.44 Example Frame

Ensure the following settings: source interval, uniformly distributed from 10 seconds to one minute (Fig. 2.45).

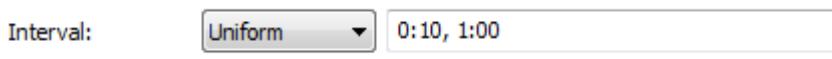


Fig. 2.45 Interval Uniform Distributed

Line: length 17 meters; speed: 0.1 m/sec. Select in the Line: Tools—Select Observers. Click on New. Prepare the settings according to Fig. 2.46.

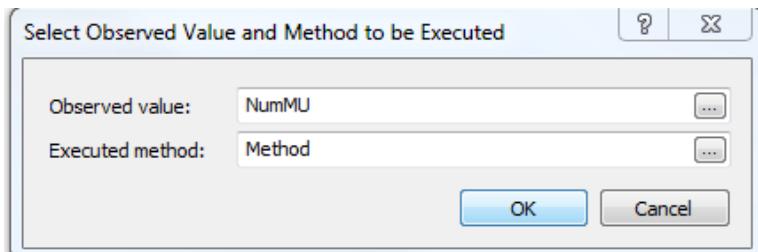


Fig. 2.46 Observer

The content of this method should resemble the following:

```
(attribute: string; oldValue: any)
is
do
  numberMUs := ?.numMu;
end;
```

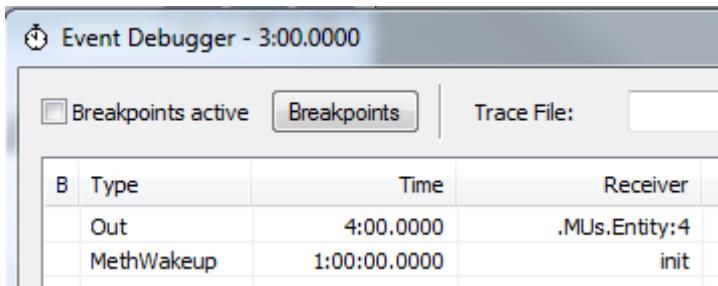
When you assign the method to the observer, this is reformatted. So you should first prepare the assignment and then program the method.

## 2.10 Interrupt Methods

In the running simulation, it may be necessary to wait e.g. for certain events and stop the processed methods during this time.

### 2.10.1 The Wait Statement

Using the wait statement, you can interrupt the execution of a method for a while, while the rest of the simulation continues. In this way, for example, you can represent distances between events that are triggered by the method. The EventController enters an event of type MethWakeUp in the event list and continues the execution of the method at the calculated time (EventController, event debugger; Fig. 2.47).



The screenshot shows the Event Debugger window with the title 'Event Debugger - 3:00.0000'. The window has a toolbar with 'Breakpoints active' and 'Breakpoints' buttons, and a 'Trace File:' input field. Below the toolbar is a table with the following data:

B	Type	Time	Receiver
	Out	4:00.0000	.MUs.Entity:4
	MethWakeUp	1:00:00.0000	init

Fig. 2.47 Event Debugger

### 2.10.2 Suspending of Methods

Frequently, you have to wait for certain events in the simulation—e.g. when a failure ends, the processing is complete, a transporter is available, etc. The method is interrupted until the condition is met. Syntax:

```
waituntil <boolean expression> prio <gZ>;
stopuntil <boolean expression> prio <gZ>;
```

The statement consists of the keywords “waituntil” or “stopuntil,” a condition followed by the keyword “prio” and an expression to calculate the priority.

### Condition

The expression specifies the conditions under which the execution of the method will be continued. If the condition is met, the interpreter continues the execution using the following statement. If the condition is not met, then the interpreter suspends the method and monitors the various elements of the expression. If a part of the condition changes later, the interpreter re-awakens the method so that the condition can be evaluated again. The structure of the expression is limited, because the interpreter must decompose the expression into observable components. The basic arithmetic operations (+, -, \*, /), comparisons (<, <=, =>, =, / =) and parentheses are permissible. Not permitted are method calls, table requests and standard methods with arguments.

### Priority, Stopuntil, Waituntil

Several methods may have been suspended due to the same or a similar condition and will also be activated together. The interpreter selects the method with the highest priority and rouses this method first. In waituntil statements, the interpreter evaluates the condition again and activates the next method if the condition is true. In stopuntil, no new evaluation of the condition before waking up the next method is carried out (even if the first method, for example, has changed the values of the condition).

### Sequence of the Suspension

The interpreter evaluates the condition of the waituntil/stopuntil instruction and suspends the method if the condition is not met. The menu command Debugger—Show suspended methods in the menu of the program window allows access to the list of suspended methods. The interpreter parses the expression in the condition and filters the observable variable proportions. If there are no such elements because the expression contains, for example, only local variables and constants, then the program will display an error message (Fig. 2.48).

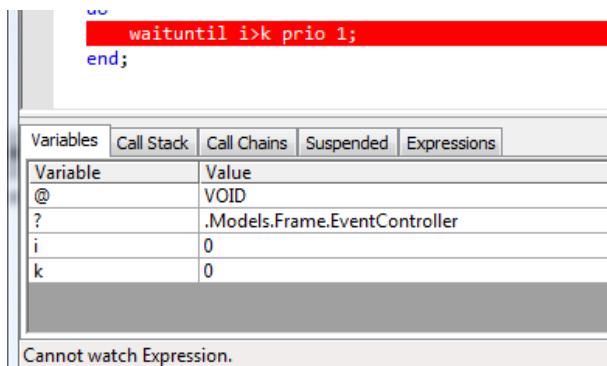


Fig. 2.48 Error Message

If the observed value changes, the interpreter activates all suspended methods that depend on this value.

### Observable Values

The condition that the continuation of a method depends on must be observable by Plant Simulation. The following values are observable:

General: values of global variables and free attributes, which have "scalar" data types (Boolean, integer, real, string, object, time, money, length, weight, speed, date, datetime). The attributes have a "\*" in the column watchable in the attribute window (Fig. 2.49).

.Models.Frame.SingleProc					
Name	Value	...	Watchable	Signature	
Failures	0			integer	
Full	false		*	boolean	
FwBlockList	0			([Forward])	

Fig. 2.49 Watchable Attribute

### Example: Stopuntil or Waituntil

Three machines load one buffer. In the buffer, only one part is to be stored. Create a frame as in Fig. 2.50.

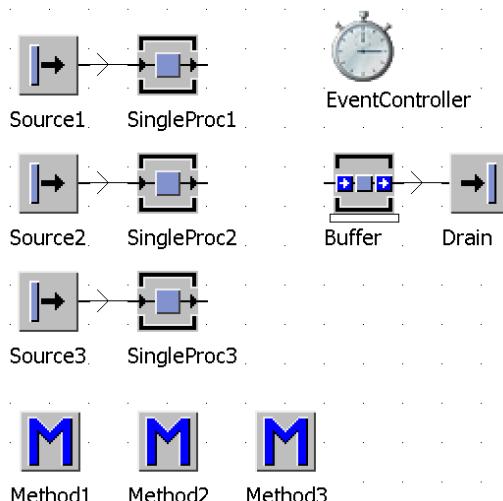


Fig. 2.50 Example Frame

Every SingleProc has 50 seconds processing time; the buffer has no processing time and a capacity of three. The sources produce parts with an interval of 50 seconds. Change the processing time of the drain to one minute (this causes a jam).

The methods are the exit controls (front) of the SingleProcs. All methods initially have the same content:

```
is
do
  waituntil buffer.empty prio 1;
  @.move(buffer);
end;
```

Let the simulation run for a while and check the statistics of the SingleProcs (tab Statistics). All stations have a working portion of approximately 28 per cent and the buffer has a maximum content of 1. As all waituntil statements have the same priority, the methods are executed sequentially. The condition is re-evaluated after each moving, so that only one part is moved (then the buffer is occupied and the condition false). Now, change the prio-value in Method1 to two. This method is now treated preferentially. The utilization of these SingleProc1 increases. No more parts are transferred from SingleProc2 and SingleProc3.

Now, change the statement waituntil in stopuntil (prio 1) and check after the simulation the statistics of the buffer. The maximum occupation is three. The interpreter awakens all the methods successively and as all methods store their part in the buffer, there is no further query of the condition buffer.empty. Thus, up to three parts are located in the buffer. You must decide case by case, which way works best for your simulation.

## 2.11 Debugging, Optimization

For searching and removing errors, several instruments are available.

### 2.11.1 *Breakpoints in Methods*

Different errors can occur when you are programming methods:

- Syntax errors: Here, you will find an error message and the method is marked as faulty if you try to store the changes. These errors are relatively easy to correct. Usually, you have a mistake in the spelling or elements of programming are forgotten or incorrectly applied (e.g. missing of the final end).
- Runtime errors: Here appears a runtime error message. Often, for example, an access takes place to the content of an empty element (e.g. a void cannot receive method xyz). Here, you need to test the system states during the simulation in order to identify the error.

- Logical errors: The simulation will initially run without error, but the behavior is wrong or the expected results are incorrect. Here you have to check the basic approach of your modeling.

In order to test the function of a method, it is possible to interrupt the execution of the function at a specific line and to control the further execution manually.

### Example: Debugging

You should program a simple counter. Create a frame as in Fig. 2.51.

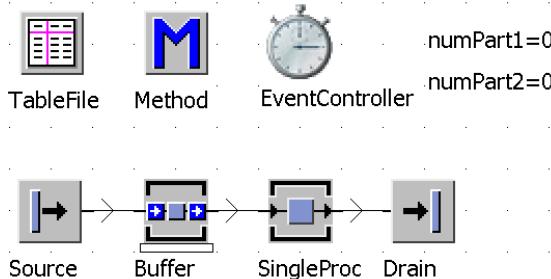


Fig. 2.51 Example Frame

The source randomly produces two types of MUs (Part1 and Part2). Create settings in the source according to Fig. 2.52.

Interval:	Uniform	0:30, 3:30	<input type="checkbox"/>
Start:	Const	0	<input type="checkbox"/>
Stop:	Const	0	<input type="checkbox"/>
MU selection:	Random	<input type="checkbox"/>	<input type="checkbox"/>
Table:	TableFile	<input type="button"/> <input type="checkbox"/>	<input type="checkbox"/> Generate as batch <input type="checkbox"/>

Fig. 2.52 Setting Source

Enter in the TableFile distributions (Fig. 2.53).

	object 1	real 2	integer 3	string 4
string	MU	Frequency	Number	Name
1	.MUs.Entity	80.00	1	Part1
2	.MUs.Entity	20.00	1	Part2

Fig. 2.53 Random Distribution

The buffer has five places and no processing time. The SingleProc has a processing time of two minutes. The method is the entrance control of the SingleProc. The method increases the counters depending on the names of the MUs. The method, therefore, has the following content:

```
is
do
  if @.name="Part1" then
    numPart1:=numPart1+1;
  else
    numPart2:=numPart2+1;
  end;
end;
```

To investigate the function of the method exactly, the method should interrupt the execution in the line with the "if". Open the method and set the cursor to line 3 and then press the F9 key (or Run—Class breakpoint). In the editor, a red dot appears at the beginning of the line. When you start the simulation, Plant Simulation opens the debugger when reaching this line (Fig. 2.54).

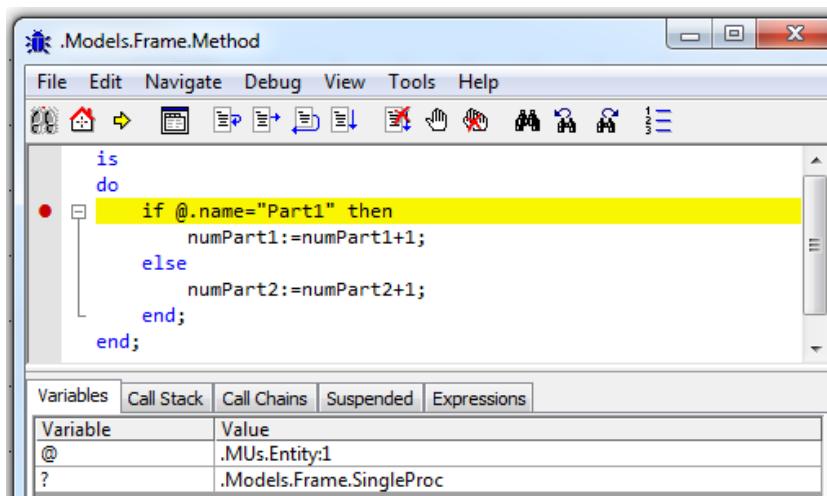


Fig. 2.54 Debugger

F11 or Debug—Step in allows you to move line by line through your program. If you point the mouse at an expression, the current value of this expression is displayed. You can also copy and paste an expression in the Expressions tab (column Expression). The debugger then determines the value of the expression and displays it. Other possibilities of continuing the program are available in the menu Debug. By observing the values of variables and values of the expressions,

you can check whether your program is working correctly. At the end of the test, you should delete the breakpoint using F9 or Debug—Class breakpoint.

### ***2.11.2 Breakpoints in the EventController***

You can also set breakpoints in the EventController. You can use the list of events to interrupt the simulation when a certain event in the simulation is taking place. Click the list button in the EventController. Select from the list the appropriate event and double-click on the corresponding row of the table. In the first column, an "S" is entered. Using Step, you can move each event through the simulation. A repeated double-click on the row deletes the breakpoint.

### ***2.11.3 Error Handler***

You have the option to replace the default error handling of Plant Simulation (opening the method debugger) through your own error handling method. You can log errors in this way and hide them from the eyes of the beholder of the simulation. This failure handling makes sense only when the simulation can run despite the error (non-critical error). For the error handling, you need to define a user-defined method attribute with the name ErrorHandler in the relevant method (Tools—User-defined Attributes—New—Name: Error Handler, Type: method). The error handler method must have three transfer parameters (Plant Simulation formats the method accordingly):

- Error (byref, string)
- Path of the method (string)
- Row (integer)

If an error occurs, Plant Simulation calls the error handler and passes the error message, the method path and the line number. If you clear the error message, Plant Simulation assumes that you have corrected the error and continues the execution of the method.

#### **Example: Errorhandler**

A just-in-sequence transport unit is to be simulated. Parts are delivered from three lines. The parts are placed in sequence on a conveyor belt. If a part is not present in the sequence, the place is left empty and filled up later.

The unsuccessful movement ("A void cannot receive repositioning method") is intended to be caught by an ErrorHandler and documented in a table with a time and sequence number. Create a frame as in Fig. 2.55.

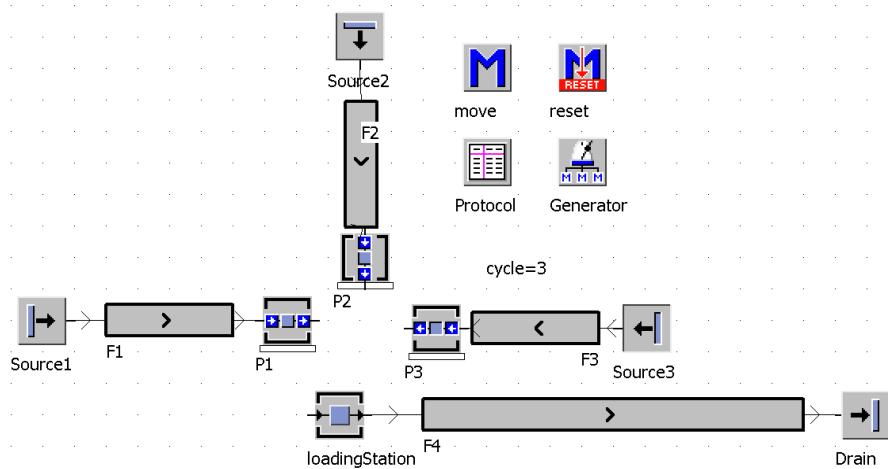


Fig. 2.55 Example Frame

Settings: Source1, Source2 and Source 3: interval of 15 seconds non-blocking; failures: 75 per cent availability; one-minute MTTR; F1 to F3 each four meters; speed of 0.0667 m/sec; LoadingStation: one second of processing time; F4 speed: 0.2 m/sec. The generator calls the method above every eight seconds. The first column of the table protocol has the data type time and the second column has the data type string.

The reset method clears the contents of the table protocol:

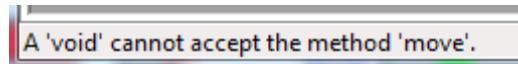
```
is
do
  protocol.delete;
end;
```

The control "move" increases the number of cycles and moves the part from the corresponding place.

```
is
do
  if cycle=1 then
    P1.cont.move/loadingStation);
  elseif cycle=2 then
    P2.cont.move/loadingStation);
  elseif cycle=3 then
    P3.cont.move/loadingStation);
  end;
  cycle:=cycle+1;
  if cycle = 4 then
    cycle:=1;
  end;
```

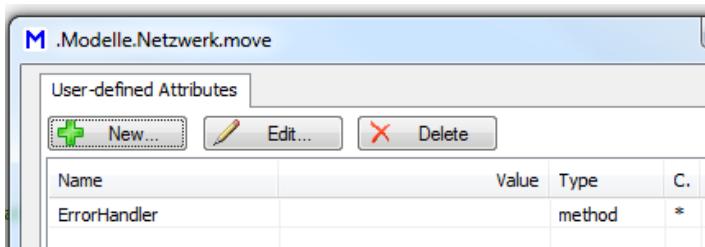
```
--if one of the P's is not occupied --> failure
--you find the errorHandler in the userdefined
-- attributes of the method
end;
```

If you run the simulation for a while, there will be an error message (Fig. 2.56).



**Fig. 2.56** Error Message

Open the method “move.” Select Tools—User-defined attributes. Add a user-defined attribute: ErrorHandler; type: method (Fig. 2.57).



**Fig. 2.57** Error Handler

Method ErrorHandler:

```
(byref failure : string; path : string;
  rowNum : integer)
is
do
  --insert in protocol
  protocol.writeRow(1, protocol.yDim+1,
    ereignisverwalter.simTime,
    "Transfer failed cycle:"+
    num_to_str(cycle));
  -- quit failure
  failure:="";
end;
```

## 2.11.4 Profiler

The profiler analyzes the runtime behavior of each individual during the simulation executed method. Hence, you can take specific steps to improve the runtime behavior of your simulation. The profiler is in the menu bar of Plant Simulation. First, activate the profiler and then run the simulation (for a certain period). Next, select Profiler—Show profile. The values of the profiler can be deleted with Profiler—Reset profiler.

### Example: Profiler

You want to test the runtime behavior of a method in different ways. Download first the example "Profiler" from [www.bangsow.de](http://www.bangsow.de). Letters should be assigned a transport zone depending on the destination (initially a destination is assigned to the letters). For this, the zone must be determined for this destination from a table. Create a frame as in Fig. 2.58. Use the table targets from the downloaded example.

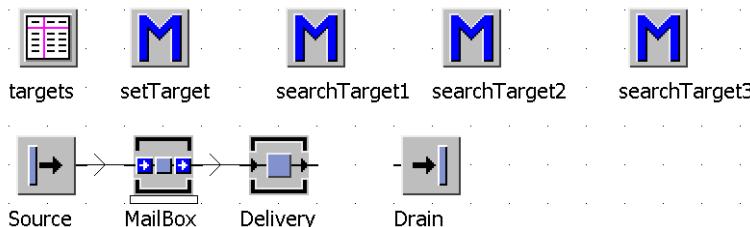


Fig. 2.58 Example Frame

Assign the method "setTarget" to the entrance control of the mailbox. SearchTarget1 is the exit control (front) of Delivery. Duplicate an entity in the class library and rename it as "Letter." Add to the letter two user-defined attributes (target: string, zone: integer). The method setTarget sets randomly a destination from the table Targets. Method setTarget:

```
is
do
  @.target:=targets[1,
    round(z_uniform(1,0,1)*targets.yDim)];
  if @.target = "destination" then
    -- correct id = 0 to id=1
    @.target:= targets[1,1];
  end;
end;
```

In the method searchTarget1, the target is to be searched for in the first column of the table targets. If the target is found, the zone is stored in the letter.

Variant1: searchTarget1

```
is
  i:integer;
do
  for i:=1 to targets.yDim loop
    if targets[1,i] = @.target then
      @.zone:=str_to_num(targets[3,i]);
    end;
  next;
  @.move(drain);
end;
```

Activate the profiler. Let the simulation run for 100 days. Then open the evaluation of the profiler (the data are dependent on the performance of your computer, Fig. 2.59).

Top 50 Most Relevant Methods					
%global	%local	seconds	calls	msec/call	name
98.6%		17.668s	143998	0.12ms	.Modelle.Netzwerk.searchTarget1
1.4%		0.254s	144099	0.00ms	.Modelle.Netzwerk.setTarget

Fig. 2.59 Profiler Variant 1

The method should be executed faster if you leave the method after a successful transfer (move letter and return). Change the method as follows:

```
is
  i:integer;
do
  for i:=1 to targets.yDim loop
    if targets[1,i] = @.target then
      @.zone:=str_to_num(targets[3,i]);
      @.move(drain);
      return;
    end;
  next;
end;
```

In this way, you will save a lot of loop iterations. Now start the Profiler and the simulation. The execution time of the method has been significantly reduced (Fig. 2.60).

%global	%local	seconds	calls	msec/call	name
96.9%		7.697s	143998	0.05ms	.Modelle.Netzwerk.searchTarget2
3.1%		0.249s	144099	0.00ms	.Modelle.Netzwerk.setTarget

Fig. 2.60 Profiler Variant 2

The TableFile object provides effective methods for searching. This will save you many unsuccessful loop iterations. Hence, the execution time of the method can be further reduced. Assign searchTarget3 to the exit control of delivery. Try the following code:

```
is
do
  targets.setCursor(1,1);
  targets.find(@.target);
  @.zone:=str_to_num(targets[3,targets.cursorY]);
  @.move(drain);
end;
```

Now, the execution time is very low (Fig. 2.61).

%global	%local	seconds	calls	msec/call	name
83.5%		1.250s	143998	0.01ms	.Modelle.Netzwerk.searchTarget3
16.5%		0.247s	144099	0.00ms	.Modelle.Netzwerk.setTarget

Fig. 2.61 Profiler Variant 3

## 2.12 Hierarchical Modeling

Often, simulations are too large to be displayed in one frame. Such simulations are divided into different frames and then combined again in a root frame.

### 2.12.1 The Frame

The frame block is the basis of all models. You can also use the frame to create your own blocks with arbitrary behavior. Therefore, you combine different basic blocks and methods to a block with higher functionality. The new block (frame) can then be used like any other block. The connection between different blocks (frames) is the interface object. Thus, it is possible to construct a model in multiple hierarchies. The block frame does not have a basic behavior. You can create new frames via the context menu of the class library folder. Click with the right mouse button on a folder icon and select New—Frame.

### 2.12.2 The Interface

The block interface allows the establishing of connections between different frames. It has only a few attributes (Fig. 2.62).

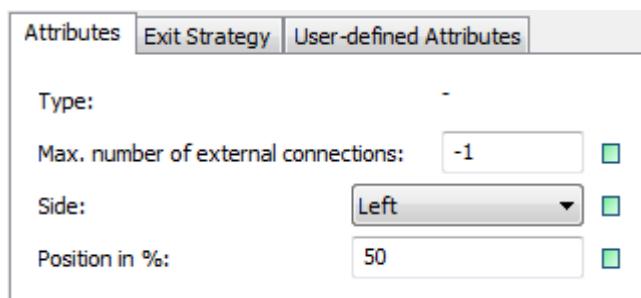


Fig. 2.62 Interface

**Type:** An interface can be either entrance or exit.

**Max. number of ... :** Determines how many predecessors and how many successors the block can have.

**Side:** Where should the interface be located on the frame icon?

**Position in %:** Specifies where the connector enters the frame-icon.

### Example: Frame and Interface

Preliminary remark: A basic concern of many object-oriented techniques is reusability. For this purpose, we use a versatile range of classes (data types). With these small building blocks (the same for all projects), we can build appropriate solutions. For scheduling, a multitude of such “kits” exists—for example, “methods time measurement (MTM)” is one. Using this methodology, human work is broken down into elementary movement objects. These movements will be assigned a time depending on various factors (time studies).

Deburring: A worker takes a part off the line. He clamps it, deburrs it, and then puts it back on the line. At work, he stands with his back to the line (Fig. 2.63).

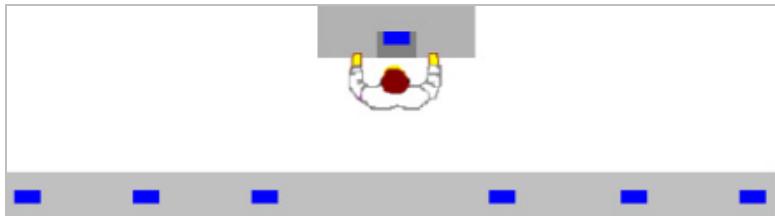


Fig. 2.63 Example

The job of the worker can, according to MTM, be broken down into the following tasks (work begins after loading the finished part onto the line)

Basic objects (abridged)	Duration (sec)
Reach (workpiece)	0.25
Grip (workpiece)	0.1
Bring	0.3
(Body) rotation (to the place)	0.5
Walk	1
Reach	0.2
Handling (clamping)	3
Reach (tool)	0.5
Grip (tool)	0.1
Bring (tool)	0.2
Handling (deburring)	3
Reach (tool)	0.5
Reach (part)	0.3
Grip	0.1
Handling (unfix the part)	2
(Body) rotation (to the line)	0.5
Walk	1
Reach	0.3
Total:	13.85 sec.

Note: To simplify matters, the element “drop” is missing. You need to create the movement objects as single classes in the class library (Fig. 2.64, download the example “Frame and Interface” from [www.bangsow.de](http://www.bangsow.de)):

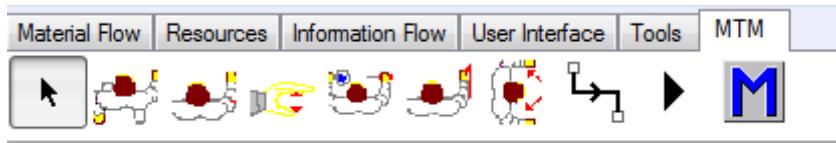


Fig. 2.64 MTM Elements

Create a frame “deburring” and insert elements as in Fig. 2.65.

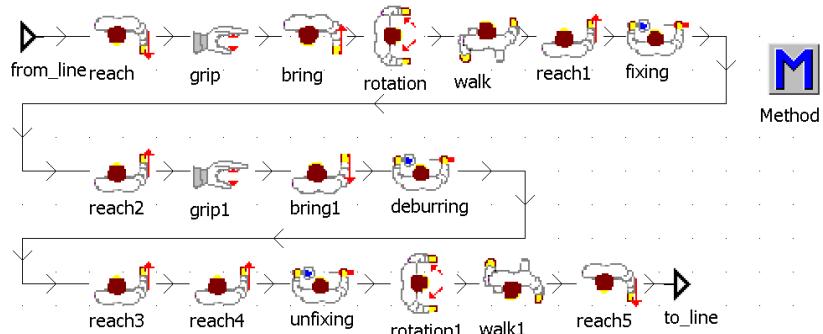


Fig. 2.65 Sub-frame Deburring

Note: In this example, you have to prevent more than one part remaining in the “deburring” frame. After an MU has entered the object “reach,” the entrance of the object “reach” must be locked and may only be opened again if the part “reach5” exits. To do this, you need a method object in the frame deburring (folder information flow). Add a method object to the frame. Open the object by double-clicking it. Turn off inheritance if you cannot enter your source code into the method editor. Enter the following source code:

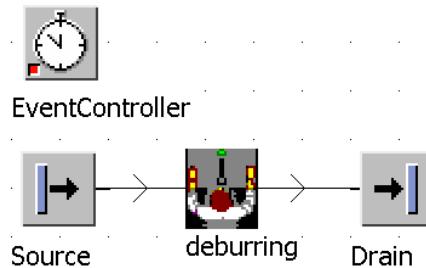
```

is
do
  if ?=reach then
    reach.entranceLocked:=true;
  else
    reach.entranceLocked:=false;
  end;
end;

```

Enter the name of the method into the object reach (tab Control—Entrance). Enter the method as exit control into the object reach5, clear the checkmark to the left of front, and select the check box rear.

You can select a new image for the frame (right mouse button—Edit icons—icon number 1). You can now use the object “deburring” in the same way as the other objects in any frame. Create a new frame and drag the frame deburring from the class library to it (Fig. 2.66).



**Fig. 2.66** Example Frame

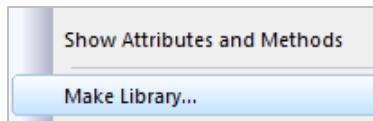
Set the interval of the source to 30 seconds. By double-clicking the object deburring you can open the sub-frame.

### 2.12.3 *Create Your Own Libraries*

You can develop your own libraries in the class library (professional license). User libraries simplify by their export and import options the re-use of model elements. When creating libraries, you must follow some rules:

- Only use elements from the library to create library items—i.e. you must duplicate all the "blocks" that you use to create the library.
- All libraries must have different names; otherwise problems might occur when importing.

The creation process is simple. Create in the class library a new folder (context menu—New—Folder) and rename it (myLib). In the context menu of the folder, choose Make Library (Fig. 2.67).



**Fig. 2.67** Make Library

A menu will appear in which you can enter information about the library. Be sure to fill out the library name field (myLib). This name is displayed as a label in the class library management.

#### Storage Location for Libraries

With Tools—Preferences—Libraries directories you set the storage location for your libraries in Plant Simulation (Fig. 2.68).

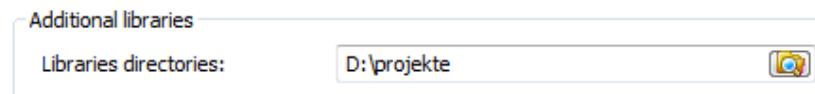


Fig. 2.68 Libraries Location

Libraries can be exported via the context menu in the class library. Select from the context menu Save/Load—Save Library as in Fig. 2.69.

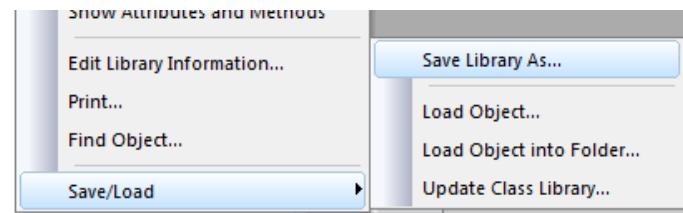


Fig. 2.69 Save Library

Save your library in the directory that you set as directories for libraries. With File—Manage Class Library, you can now import your library into another model. You find the user libraries at the end of the tab Libraries (Fig. 2.70).



Fig. 2.70 User Libraries

## 2.13 Dynamic Model Generation

With the help of SimTalk, you are able to create dynamic models with a manageable amount of programming. The data can come from different sources. Shown is the generation based on data that are stored using a simple import from Excel into Plant Simulation.

### 2.13.1 Required Data, SimTalk Language Elements

In the simplest form, you need to generate a model of the following data:

- Model elements (type, name)
- Positions (x, y coordinate)
- Relationship between the elements (predecessor—successor)
- Element attributes (processing time, failure data, setup behavior)
- Settings of the sources

Table 2.9 lists the SimTalk language elements for the dynamic generation of a model.

**Table 2.9** Model Generation, SimTalk

Method /Attribute	Description
<path>.createObject(object destination, integer x, integer y, string name)	Creates an instance of the class specified with <path> at the position (x, y) and renames it with name
<connector-path>.connect(object start, object end)	Connects the objects' start and end with a connector

As an example, you should generate a simple model consisting of four machines, a source and a drain. To do this, enter the information from Fig. 2.71 to a table Data in an otherwise empty frame.

	string 1	string 2	string 3	real 4	real 5	real 6
string	Name	Object type	Successor	x	y	Processing time
1	Source	Source	2,3	10.00	10.00	120.00
2	Machine1	SingleProc	4	20.00	10.00	240.00
3	Machine2	SingleProc	4	20.00	20.00	240.00
4	Machine3	SingleProc	5	30.00	15.00	120.00
5	Machine4	SingleProc	6	40.00	15.00	120.00
6	Drain	Drain		50.00	15.00	0.00

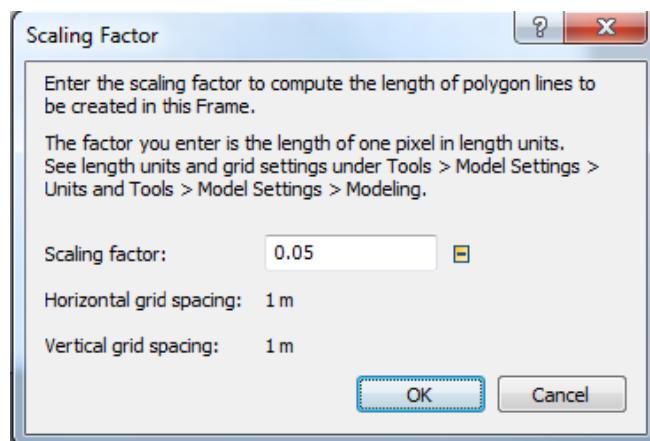
**Fig. 2.71** Table Data

### 2.13.2 *Create Objects and Set Attributes*

Before you can create objects, you must first set the scaling of your frame. The scaling determines the mapping relationship between pixels and meters.

In the basic setting of Plant Simulation, a pixel corresponds to 0.05 m or 20 pixels (default grid spacing) correspond to one meter. You set the scaling factor in the frame with Tools—Scaling factor (Fig. 2.72).

The position 0,0 is in the upper left corner of the frame. The positioning of the individual objects is carried out at their reference points. Take into account the distance between the reference point and the edge of the icon for calculating



**Fig. 2.72** Scaling factor

the pixel position. The scaling factor of the frame you can read with `<frame>.scalingFactor`. A method for the generation of all objects in the data table at the corresponding position would look like this:

```

is
  obj,obj_class:object;
  x,y,i:integer;
do
  for i:=1 to data.yDim loop
    --create class from data
    obj_class:=str_to_obj(".Materialflow."
      +Data[2,i]);
    -- calculate pixel from meter
    x:=round(Data[4,i]/root.scalingFactor);
    y:=round(Data[5,i]/root.scalingFactor);
    --create and rename objects
    obj:=obj_class.createObject(root,x,y,Data[1,i]);
  next;
end;

```

After generating the objects, you can use the returned reference to set attributes, such as the processing time. In the example above, note that the time in the source is set with `<source>.interval`. You can use the class name (`<path>.class.name`) for differentiating the objects. An extension of the method (inside the loop) to set the processing times could be as follows:

```

-- set proctimes of the objects
if obj.class.name="Source" then
  obj.interval:=num_to_time(Data[6,i]);
else
  obj.procTime:=num_to_time(Data[6,i]);
end;

```

### 2.12.3 *Link Objects Dynamically with Connectors*

To link the objects with connectors you need the relationships between the objects (successors). As in this example, an object may have several successors. A common way of defining is by using the line number in the object list as a reference. When you import data from other applications, you will pass these successors usually as enumeration (a text with delimiter). In preparation, you first need a way to split the enumeration into a list of integer values.

Note: Starting from Version 11 TR3 Plant Simulation provides such a method by default: `splitString`.

You find a user-defined split method in: 2.9.4

The connection of the elements requires the following steps:

1. All successors (row numbers) will be determined and included into a list (split function).
2. Using the method `str_to_obj`, object references will be generated from the first column of the table data and passed to the method `<connector>.connect`.

Therefore, the following program is necessary:

```

is
  obj,obj_class:object;
  x,y,i,k:integer;
  succList:list;
do
  for i:=1 to data.yDim loop
    --create class from data
  next;

  --set connectors
  for i:=1 to data.yDim loop
    if data[3,i] /= void then
      -- look for the row numbers of the successors
      succList:=split(data[3,i],",");
      -- connect with all successors
      for k:=1 to succList.dim loop
        .Materialflow.connector.connect(
          str_to_obj(data[1,i]),
          str_to_obj(data[1,
            str_to_num(succList.read(k))]));
      next;
    end;
  next;
end;

```

### 2.13.4 Connect Objects Dynamically with Lines

If the stations have to be connected by conveyor lines, you must first create the conveyor lines and then link stations and conveyors with connectors. The generation of the conveyor lines (tracks) is carried out in two steps:

1. The first step is the insertion of the conveyor line into the frame. The starting point is set to the reference point of the first object (or somewhere beside it). First, the conveyor line has a default length and a default orientation (left to right). Data about the path of the conveyor line (path segments) are stored in the segments table.
2. As a second step, the necessary length and orientation of the conveyor line are generated by changing the end point in the segments table.

For the dynamic generation of the conveyor lines, the commands from Table 2.10 are required.

**Table 2.10** Model generation, commands

SimTalk command	Description
<path>.xPos	x-position of the object
<path>.yPos	y-position of the object
<path>.segmentsTable	Reads and sets the segments table of the object

To change the curve of the conveyor line, you must first read the segments table, change the values and assign the changed table once again to the line. After assigning the changed segments table, Plant Simulation calculates the length of the line anew if you have enabled the option "transfer length" in the tab curve. In the segments table, change the first and second values in the second row (x- and y-coordinates of the end point; see Fig. 2.73).

Index	Anchor point X	Anchor point Y	Anchor point Z	Angle	Length
0	160.000000	200.000000	20.000000	0.000000	40.000000
1	200.000000	200.000000	20.000000	0.000000	0.000000

**Fig. 2.73** Segments Table

The connecting of the objects with conveyor lines requires the following steps:

1. All successors (row numbers in table data) are determined and included into a list (function split).
2. Using the method str\_to\_obj, object references are generated for predecessors and successors from the first column of the table Data.

3. At the position of the predecessor, a conveyor line is generated.
4. The segments table of a conveyor line is read. The end of the conveyor line is set to the position of the successor. The modified segments table is reassigned to the conveyor line.
5. Predecessors, conveyor lines and successors are linked with connectors.

This results in the following program:

```

is
  obj,obj_class:object;
  x,y,i,k:integer;
  succList:list;
  succ_object:object;
  pred_object:object;
  line:object;
  segTab:table;
do
  for i:=1 to data.yDim loop
    --create class from data
    obj_class:=str_to_obj(".Materialflow."+Data[2,i]);
    -- calculate pixel from meter
    x:=round(Data[4,i]/root.scalingFactor);
    y:=round(Data[5,i]/root.scalingFactor);
    --create and rename objects
    obj:=obj_class.createObject(root,x,y,Data[1,i]);
    -- set proctimes of the objects
    if obj.class.name="Source" then
      obj.interval:=num_to_time(Data[6,i]);
    else
      obj.procTime:=num_to_time(Data[6,i]);
    end;
  next;
  --set lines and connectors
  for i:=1 to data.yDim loop
    if data[3,i] /= void then
      pred_object:=str_to_obj(data[1,i]);
      -- look for the row numbers of the successors
      succList:=split(data[3,i],",");
      -- connect with all successors
      for k:=1 to succList.dim loop
        succ_object:=
        data[1,str_to_num(succList.read(k))];
        -- create line
        line:=.MaterialFlow.Line.createObject(
          root,pred_object.XPos,
          pred_object.YPos);

```

```
--read segments table
segTab:=line.SegmentsTable;
--set end coordinate
segTab[1,2]:=succ_object.XPos;
segTab[2,2]:=succ_object.YPos;
--re-assign segments table
line.SegmentsTable:=segTab;
--link all together
.materialFlow.connector.connect(
    pred_object,line);
.materialFlow.connector.connect(
    line, succ_object);
next;
end;
next;
end;
```

## 2.14 Dialogs

You can use the object dialog to create your own dialog boxes. You can create dialogs for your own objects for the user of the simulation model to facilitate the operation of your objects. The dialog can be used to select the settings in complex frames and sub-frames. It serves as an interface between the simulation and the user. In this way, you can create simulations that can be operated by users who have no knowledge of Plant Simulation.

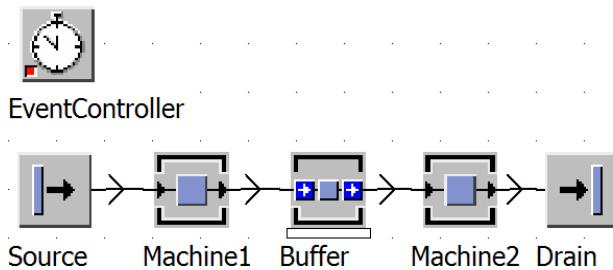
### 2.14.1 *Elements of the Dialog*

Each dialog object manages a single dialog box. A dialog may consist of the following basic elements:

- Comments/labels
- Text fields
- Buttons, menus
- List-boxes
- Radio buttons and checkboxes
- Tabs
- Images
- Tables

#### **Example: Dialog**

The processing time and the failures of two machines are to be set via a dialog object. Create a frame according to Fig. 2.74.



**Fig. 2.74** Example Frame

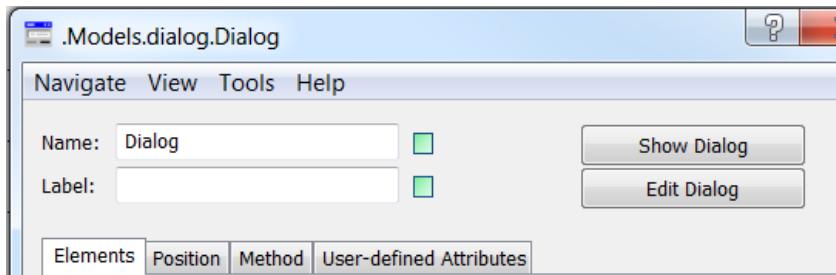
The dialog seeks to control the following settings:

Object	Attribute
Machine1	ProcTime, failed
Machine2	ProcTime, failed
Buffer	Capacity

Initially set the processing time of the machines to one minute. Insert a dialog object into the frame.

### 2.14.1.1 The Dialog Object

Double clicking or selecting Open on the context menu opens the dialog object (Fig. 2.75).



**Fig. 2.75** Dialog Window

Clicking the button Show Dialog allows you to view the dialog. Clicking Edit Dialog opens a dialog editor. You can arrange the individual elements of the dialog on the tab Elements and determine the position of the dialog box when it is called on the tab Position.

#### Insert Elements

Static text boxes explain the dialog. Use static text boxes as labels of the input text boxes and for general operating instructions. Select the item you want to insert on the context menu on the tab Elements.

Example: To enter the processing time of Machine1, you need a static text box (as identifier) and an edit text box (to enter text).

1. Click the right mouse button on the tab Elements, and select New Static Text Box. Fill the dialog box as in Fig. 2.76.

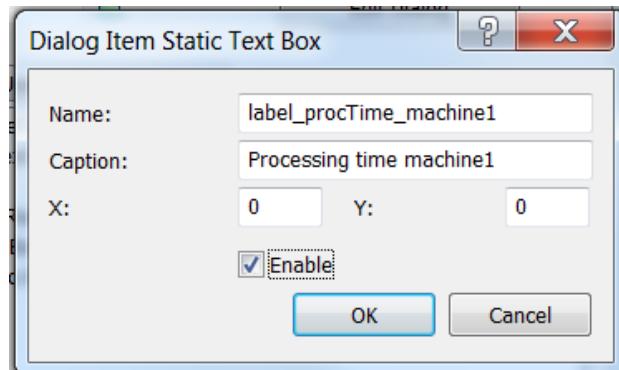


Fig. 2.76 Static Text Box

The name is the address of the text box. The caption is the text displayed on the dialog box. X and Y are the positions of the element in the dialog (column, row). The positions start at X = 0, Y = 0 (top left). You can set this quite easily afterward by clicking the Edit Dialog button. Then, drag the fields to the correct position.

2. Click again with the right mouse button on the tab Elements. Select the New Edit Text box on the context menu and enter the data as in Fig. 2.77.

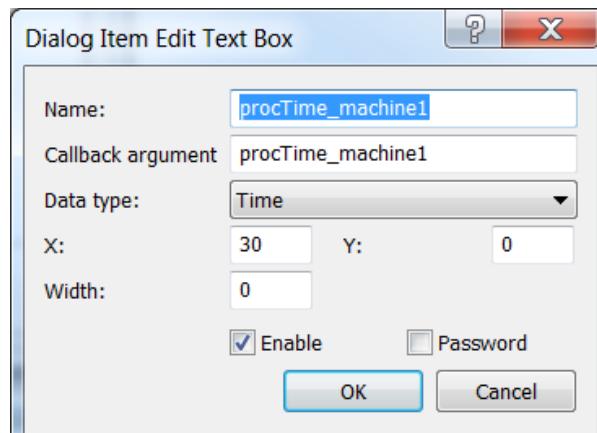


Fig. 2.77 Edit Text Box

The callback argument allows you to access the field. The setting in the data type field restricts the possibilities of user input. If you clear the checkmark at Enable, then the item is disabled and no user input is accepted. If you activate the option Password, then the entries in this field will be shown masked. You can double-click each element in the tab Elements to open and edit it. Click the button Show Dialog (Fig. 2.78).

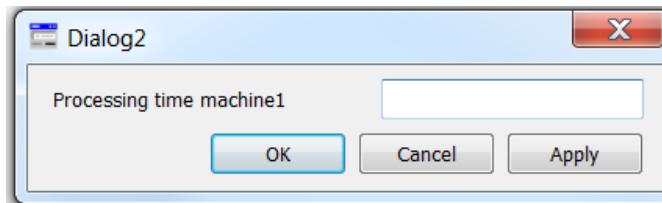


Fig. 2.78 Dialog

#### 2.14.1.2 Callback Function

When you click OK, Cancel, or Apply, the dialog calls a method and passes a value (the callback argument), which makes it possible to recognize which button the user has clicked. The method is defined in the tab Method (default: self.callback). You can open the method by pressing F2. There are three predefined arguments (Fig. 2.79):

- Open
- Close
- Apply

OK calls the callback function twice; Apply and Close call it once each.

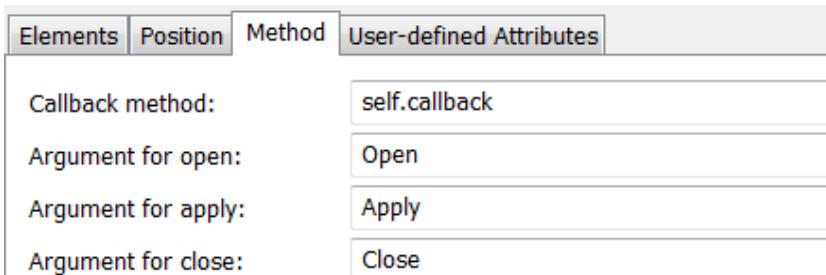


Fig. 2.79 Dialog Callback Method

Within the callback function, you need to program what will happen when the user clicks on the respective buttons.

Example: When the user clicks the buttons in the dialog, the console should display a relevant message (to demonstrate). The callback function should have the following form:

```
(action : string)
is
do
  inspect action
  when "Open" then
    print "Dialog open";
  when "Apply" then
    print "Apply clicked";
  when "Close" then
    print "OK or Cancel clicked";
  end;
end;
```

### 2.14.1.3 The Static Text Box

Static text boxes are needed to display text in the dialog. You can modify the contents of the static text box at runtime. Use the method `<path>.setCaption(<string1>, <string2>)`. The method `setCaption` sets the caption of the dialog element `<string1>` to the text `<string2>`. Example (Method callback): When you click Apply, the text of the static text box changes:

```
(action : string)
is
do
  inspect action
  when "Open" then
    print "Dialog open";
  when "Apply" then
    print "Apply clicked";
  when "Close" then
    print "OK or Cancel clicked";
  end;
end;
```

### 2.14.1.4 The Edit Text Box

The user can enter text into edit text boxes. Table 2.11 contains important methods.

**Table 2.11** SimTalk methods for accessing Edit Text Boxes

Method	Description
<code>&lt;path&gt;.setCaption(&lt;string1&gt;, &lt;string2&gt;)</code>	Set the contents of the text box <code>&lt;string1&gt;</code> to the new contents <code>&lt;string2&gt;</code>
<code>&lt;path&gt;.getValue(&lt;string1&gt;)</code>	Returns the contents of the text box <code>&lt;string1&gt;</code> (data type text).
<code>&lt;path&gt;.setSensitive(&lt;string1&gt;, &lt;Boolean&gt;)</code>	Activates/deactivates the element <code>&lt;string1&gt;</code>

Example: When opening the dialog, the edit text box should show the current processing time of the Machine1. Clicking Apply should set the processing time of Machine1 anew. For the next step, you need conversion functions: str\_to\_time(<string>) to convert text to the data type time, and to\_str(<any>) to output any data type as text. Callback function:

```
(action : string)
is
do
  inspect action
  when "Open" then
    --enter the processing time of Machine1
    --into the text box
    dialog.setCaption("procTime_machine1",
      to_str(machine1.procTime));
  when "Apply" then --new procTime machine1
    machine1.procTime:=
      str_to_time(dialog.getValue("procTime_machine1"));
  when "Close" then
    end;
end;
```

### 2.14.1.5 Images in Dialogs

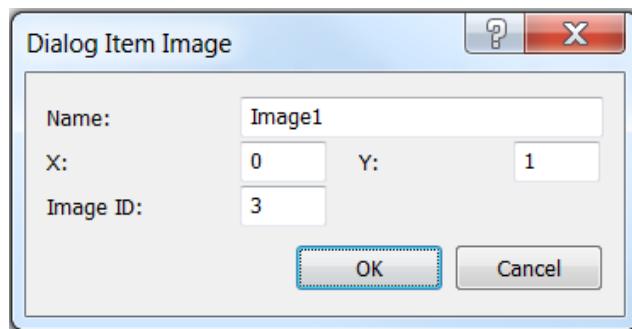
You can display images in the dialog that you have previously defined in the icon editor as an icon of the dialog (Context menu—Edit Icons ...).

Example: Insert a new icon in the report in the icon editor. Use an icon from the icon library (Tools—Load Icon, V12: Edit—Import—Import Icon Resource, Fig. 2.80).



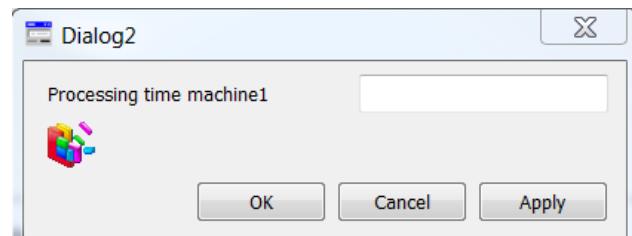
Fig. 2.80 Dialog Icon

Select New Image from the context menu on the tab Elements. Enter the icon with the number from the icon editor (3 in our example, Fig. 2.81).



**Fig. 2.81** Dialog Item Image

The image is displayed in the second row ( $Y = 1$ ) in the first column ( $X = 0$ ), Fig. 2.82.



**Fig. 2.82** Dialog

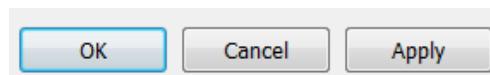
Plant Simulation provides some methods for manipulating the images on the dialogs (Table 2.12).

**Table 2.12** SimTalk methods for accessing images

Method	Description
<code>&lt;path&gt;.setIcon(&lt;string1&gt;, &lt;string2&gt;)</code>	Sets the image id of the image <code>&lt;string1&gt;</code> to <code>&lt;string2&gt;</code>
<code>&lt;path&gt;.getIcon(&lt;string1&gt;)</code>	Returns the image id of the image <code>&lt;string1&gt;</code> as string

#### 2.14.1.6 Buttons

If the option Show Default Buttons on the tab Elements is selected, the dialog is displayed with three standard buttons (Fig. 2.83).



**Fig. 2.83** Dialog Buttons

You can also create a dialog with your own buttons.

### Example: Error Dialog

Let us say that you want to design your own message window. It will contain an error message and a symbol, as well as an OK button to close the window. Create a dialog (Error\_Message), and insert an image (error\_image) and a static text box (message), Fig. 2.84.

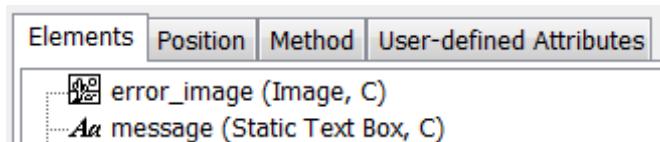


Fig. 2.84 Dialog Example

Clear the checkbox Show Default Buttons. Then add a button on the tab Elements. Enter settings according to Fig. 2.85.

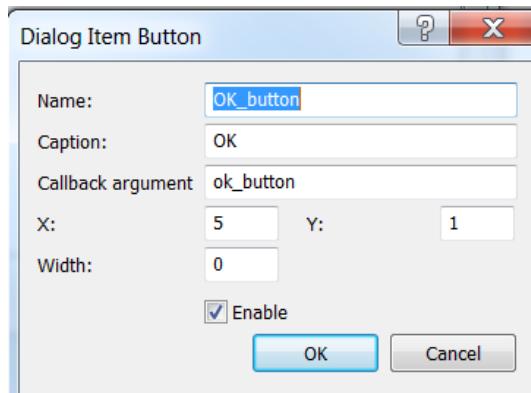


Fig. 2.85 Dialog Button

You have to program in the callback function the closing of the window. It could appear as follows:

```
(action : string)
is
do
    inspect action
    when "ok_button" then
        error_Message.close(false);
    end;
end;
```

For buttons on dialogs, Plant Simulation provides the following methods (Table 2.13):

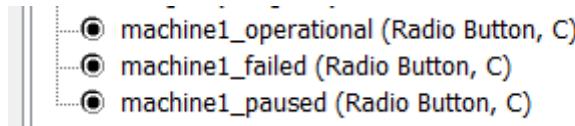
**Table 2.13** SimTalk Methods for Buttons

Method	Description
<path>.setCaption(<string1>, <string2>)	Sets the caption of the button with the name <string1> on <string2>
<path>.setSensitive( <string1>,<Boolean>)	Activates /deactivates the button <string1>

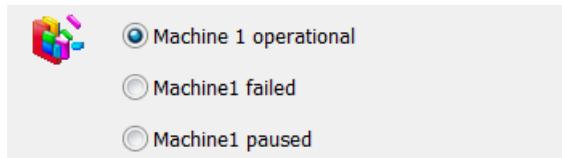
### 2.14.1.7 Radio Buttons

Radio buttons can represent only two values (true and false). Multiple option fields allow selection from a given set of options. A new selection deselects the previous selection. The assignment of the radio buttons to each other is set with a GroupID (integer). All fields with the same GroupID belong together.

Example: For Machine1, different states should be offered: operational, paused, and failed. The radio buttons have their own label (caption). Expand the dialog of the example Dialog of three radio buttons (Fig. 2.86).

**Fig. 2.86** Radio Buttons

The settings of the radio buttons are: GroupID = 0, callback arguments: machine1\_operational, machine1\_failed, Machine1\_paused, labels according to Fig. 2.87.

**Fig. 2.87** Radio Buttons (Labels)

You must set up an area in the branch Apply. Set up the callback function for each radio button in which you specify what is to happen if the option is selected. If you set up the callback arguments directly as a query (e.g. when “machine1\_failed” then ...) in the callback function, the change will take place without the user having clicked Apply or OK (clicking the radio buttons calls the callback function). The method

```
<path>.getCheckBox(<string>)
```

returns which radio button was selected. The return value has the data type Boolean, method self.callback:

```

(action : string)
is
do
  inspect action
  when "Open" then
  when "Apply" then --new procTime machine1
    --get state of radio buttons and set state
    --of machine1
    machine1.failed:=
      dialog.getCheckBox( "machine1_failed" );
    machine1.pause:=
      dialog.getCheckBox( "machine1_paused" );
  when "Close" then
  end;
end;

```

When opening it, the dialog is to display the state of machine1. Set the value of the radio box with the method

```
<path>.setCheckBox(<string>, <boolean>);
```

Example: In the callback function above, expand the branch for Open as follows:

```

when "Open" then
  dialog.setCheckBox( "machine1_failed",
    machine1.failed );
  dialog.setCheckBox( "machine1_paused",
    machine1.pause );
  if not machine1.failed and not
    machine1.pause then
    dialog.setCheckBox( "machine1_operational", true );
  end;
when "Apply" then

```

The radio button provides the following methods (Table 2.14):

**Table 2.14** SimTalk Methods for Radio Buttons

Method	Description
<path>.setCaption(<string1>, <string2>)	Sets the caption of the radio button with the name <string1> on <string2>
<path>.setSensitive( <string1>, <Boolean> )	Activates/deactivates the radio button <string1>
<path>.setCheckBox(<string>, <boolean>);	Sets the status <boolean> of element <string>
<path>.getCheckBox(<string>)	Returns the status of the element <string>

### 2.14.1.8 Checkbox

The Checkbox can be either selected or cleared. It represents two states (e.g., paused, not paused) and provides the same methods as the radio button.

### 2.14.1.9 Drop-Down List Box and List Box

If you want to provide a wide range of choices, then radio buttons require too much room in the dialog. For such cases, you can use list boxes (there are always some items visible in a list) or drop-down list boxes (there is only one entry visible; only for a selection, the list will be expanded).

#### Example: Dialog Continuation

A dialog displays the statistical data of different objects. Add a dialog to the frame and name it `statistics_dialog`. Add elements to the dialog like in Fig. 2.88.



Fig. 2.88 Dialog Elements

Captions (Fig. 2.89):

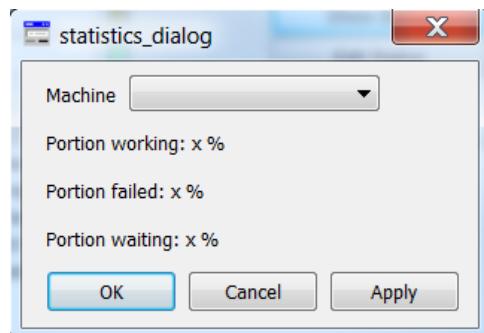


Fig. 2.89 Dialog Captions

You can easily enter the values of the drop-down list into a table (click the button `Items` in the properties dialog, Fig. 2.90).



Fig. 2.90 List Box Items

You can identify the selected entry with the method `<path>.getValue(<string>)` (pass the name of the drop-down list box). To access the corresponding objects (e.g. machine), convert the string into an object (`str_to_obj(<string>)`).

An evaluation (e.g. after clicking the Apply button, or selecting another entry in the drop-down list) might appear as follows:

```
(action : string)
is
  mach:object;
do
  inspect action
  when "Open" then
  when "Apply", "maschine" then
    --get entry and read statistical data
    mach:=str_to_obj(statistics_dialog.getValue(
      "machine_selection"));
    --set captions of static text boxes
    statistics_dialog.setCaption("text_working",
      "Portion working: " +
      to_str(round(mach.statWorkingPortion*100,2)) +
      " %");
    statistics_dialog.setCaption("text_failed",
      "Portion failed: " +
      to_str(round(mach.statFailPortion*100)) + " %");
    statistics_dialog.setCaption("text_waiting",
      "Portion waiting: " +
      to_str(round(mach.statWaitingPortion*100)) +
      " %");
  when "Close" then
  end;
end;
```

Additional methods of the drop-down list box are (Table 2.15):

**Table 2.15** SimTalk Methods of List Box

Method	Description
<code>&lt;dialog&gt;.setIndex(&lt;string1&gt;, &lt;integer&gt;)</code>	Selects the entry with the index <code>&lt;integer&gt;</code> in the dialog element with the name <code>&lt;string1&gt;</code>
<code>&lt;dialog&gt;.getIndex(&lt;string&gt;)</code>	Determines the index of the selected list entry

The functionality of the list box is similar to the drop-down field, but multiple list entries are always visible.

#### 2.14.1.10 List View

The list view displays the contents of a table in a dialog. The user can select a row from the table (the ListView returns the number of the selected row).

### Example: Dialog Continuation

A company manufactures three parts in a product mix. You are to simulate various mixes and test a selection of mixes (a certain amount of part1, part2, and part3 each). The proportion of part1, part2, and part3 is 1:3:5. First, create three entities (part1, part2, part3) in the class library. Create a table “mix” in the frame according to Fig. 2.91.

	string 0	integer 1	integer 2	integer 3	str 4
string		part1	part2	part3	
1	mix1	1	3	5	
2	mix2	5	15	25	
3	mix3	10	30	50	
4	mix4	20	60	100	
5	mix5	100	300	500	
-					

Fig. 2.91 Table Mix

Place a dialog (mix\_choice) into the frame. Insert a static text box (as title) and a list view (mixtable), as in Fig. 2.92.

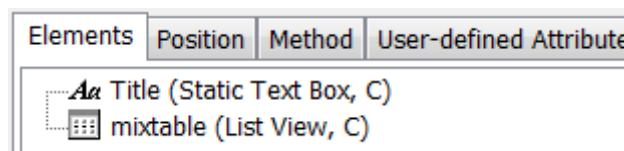


Fig. 2.92 Dialog mix\_choice

Make settings like in Fig. 2.93 for the list view.

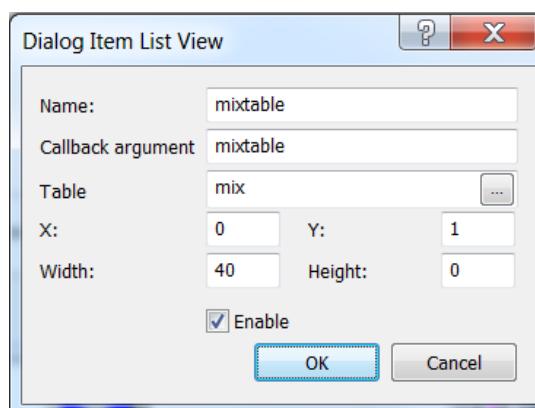
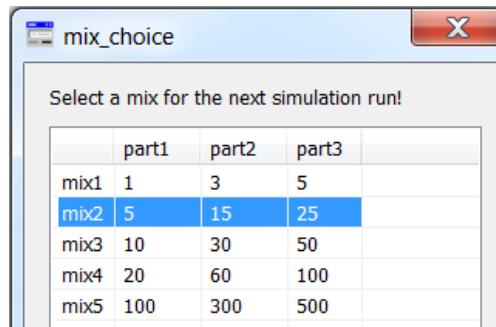


Fig. 2.93 List View Configuration

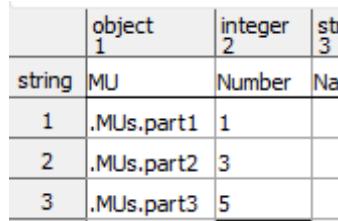
The content of the table is now displayed in the dialog (Fig. 2.94).



	part1	part2	part3	
mix1	1	3	5	
mix2	5	15	25	
mix3	10	30	50	
mix4	20	60	100	
mix5	100	300	500	

Fig. 2.94 Dialog with List View

Example: You need a production table (production, Fig. 2.95) for the source (set the source to MU selection: sequence cyclical).



	object 1	integer 2	st 3
string	MU	Number	Na
1	.MUs.part1	1	
2	.MUs.part2	3	
3	.MUs.part3	5	

Fig. 2.95 Table Production

When the user clicks Apply or OK, the number of parts from the selected mix in the rows of the table production will be transferred. After that, the source produces the new mix. Callback function:

```
(action : string)
is
  row:integer;
do
  inspect action
  when "Open" then
  when "Apply" then
    -- TODO: add code for the "Apply" action here
    -- read index of selected row
    row:=mix_choice.getRow("mixtable");
    -- write selected product mix
    production[2,1]:=mix[1,row];
    production[2,2]:=mix[2,row];
    production[2,3]:=mix[3,row];
  when "Close" then
  end;
end;
```

**Table 2.16** SimTalk Methods of the List View

Method	Description
<path>.setTable(<string>, <object>)	Sets the table of the element <string> to <object>
<path>.setSensitive( <string1>,<Boolean>)	Activates/deactivates the radio button <string1>
<path>.setTableRow(<string>, <integer>)	Sets the selected row in the ListView
<path>.getTable(<string>)	Returns the table name of the element <string>
<path>.getTableRow(<string>)	Returns the index of selected row (starting from 1)

### 2.14.1.11 Tab Control

If you have to arrange a number of elements on your dialog, presenting them on several tabs is helpful. All elements belonging to a topic are grouped on one tab. You first have to create a tab control and then add tabs to this element. New items are always created on the tab on which they are to be shown (context menu of the tab). The dialog structure is shown as a tree.

### 2.14.1.12 Group Box

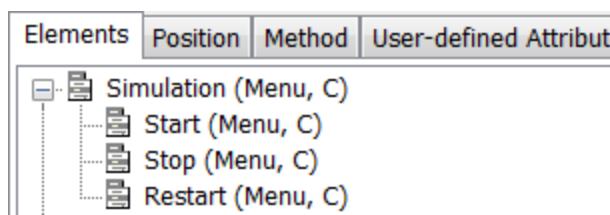
The group box graphically groups elements. Create the elements within the group box using the context menu of the group box. The group box has a separate address (elements in the group start again at field positions x = 0, y = 0). Elements of the group are displayed in the tree below the group box.

### 2.14.1.13 Menu and Menu Item

Menu items work just like buttons. When the user clicks a menu item, Plant Simulation passes the callback argument to the callback function. Within it you can evaluate the callback argument and initiate the necessary action.

#### Example: Dialog Continuation

You are to control the simulation with a dialog. It is to be a menu in the dialog with the menu items: Start, Stop, Restart (stop, reset, init, start). Name the callback arguments the same as the menu items. First, insert the menu Simulation. Then create the menu items using the context menu of the menu Simulation (Fig. 2.96).

**Fig. 2.96** Dialog Menu

Callback method:

```
(action : string)
is
do
    inspect action
    when "Start" then
        eventController.start;
    when "Stop" then
        eventController.stop;
    when "Restart" then
        eventController.stop;
        eventController.reset;
        eventController.init;
        eventController.start;
    end;
end;
```

**Table 2.17** SimTalk Methods for the Menu

Method	Description
<path>.setCaption(<string1>, <string2>)	Sets the caption of the menu/menu item with the name <string1> on <string2>.
<path>.setSensitive (<string1>,<Boolean>)	Activates/deactivates the menu/menu item <string1>

## 2.14.2 Accessing Dialogs

Important methods to access the dialogs are shown in Table 2.18.

**Table 2.18** SimTalk Methods for Accessing Dialogs

Method	Description
<path>.open	Shows the dialog
<path>.openDialog	Opens the dialog of the dialog
<path>.close	Hides the dialog
<path>.closeDialog	Closes the dialog window

When you open the frame, the dialog is to be shown automatically. The method must first show the frame and then the dialog window.

```
is
do
    .models.frame.openDialog;
    dialog.open;
end;
```

The method will be allocated in the frame with: Tools > Select Controls (V. 12 Ribbons: Home—Objects—Controls). Here, you can define different actions and how they are triggered. The method is executed once the frame is opened (Open).

### 2.14.3 User Interface Controls

Starting with Version 10, you can insert controls directly into the frame. Available at present are:

- Checkbox
- Drop-down list boxes
- Button

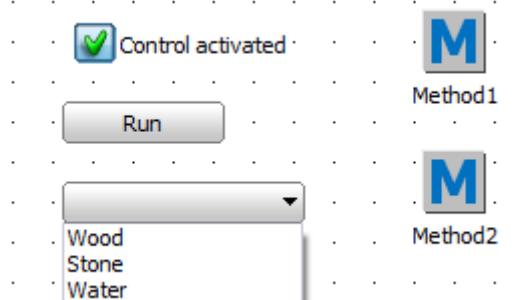
You can find the controls in the folder "UserInterface" of the class library. A method is called when you click (select) the control. In the method, you can access relatively easily the selected value and trigger appropriate actions. You need the following SimTalk methods (Table 2.19):

**Table 2.19** SimTalk methods of the user interface controls

Method	Description
<path>.value	Returns the value of the checkbox (boolean) or the number of the selected item of the listbox (integer)
<path>.item	Return the selected item of the listbox as a string
<path>.items	Returns all items of the listbox as an array

#### Example: Interface Controls

Create a simple frame according to Fig. 2.97.



**Fig. 2.97** Example Frame

Since the action of the controls is triggered with a mouse click, use the context menu (right-click—Open) to display the menu of the objects. The configuration of the elements is relatively simple. The text that you enter in the field Label is displayed as the label of the element. You can specify the default value (state) of the object with value (checkbox). In the drop-down list, you can specify the entries via the button Items. The method assigned to Control is called when you press/select the control

element. To test this, assign method1 to the checkbox and method2 to the drop-down list. The control of the drop-down list is called only when the user selects an item in the list. The control elements can be accessed with "?" in the methods. Method1 (reading of the checkbox):

```
is
do
  if ?.value = true then
    messageBox("selected",1,14);
  else
    messagebox("not selected",1,14);
  end;
end;
```

Method2 (reading the value of the drop-down listbox)

```
is
do
  messagebox(? .item,1,14);
end;
```

## 2.14.4 *Input Functions*

There are a number of predefined dialogs for entering data. They will be called as a function, and will return a value after closing by the user. The following input functions are available (Table 2.20):

**Table 2.20** Input functions

Method	Description
prompt (<string>,[<string>])	Shows a simple command prompt and the supplied text is displayed as a prompt. You can enter text and it returns a string.
promptList1(<list>,[<string>])	You will see a selection list. You can select exactly one element. You pass a list of entries and text for the caption of the menu. Returned is an integer value of the selected item or zero if Cancel is clicked.
promptList1N(<list>,[<string>])	You will see a selection list. You can select multiple items (press the Shift or Ctrl key). You pass a list of entries and text for the caption of the menu. The method returns a list of integer values of the selected entries or zero if Cancel is clicked.
openColorSelectBox (<integer>)	Opens a color selection menu; as a parameter, you pass a default color (e.g. as a result of makeRGBValue).
openObjectSelectBox(<string>,<object>);	Opens an object selection window. As the first parameter you pass an object filter (" or ":"object "), while the second parameter is the starting point for the display
selectFileForSave / selectFileForOpen	Shows a File Save As or Open menu (see Help for parameters)

**Example: Input Functions**

Add a new frame. Insert a method-block into the frame. Check the following program:

```
is
    retVal:string;
    values:list[string];
    entry:integer;
    entries:list[integer];
    i:integer;
    txt:string;
do
    --prompt function
    retVal:=prompt("This is the prompt input function",
        "Enter a value and press OK");
    messageBox(retVal, 1, 4);
    --create list and fill it with values
    values.create;
    values.append("Wood");
    values.append("Glas");
    values.append("Air");
    --selectbox
    entry:=promptList1(values,
        "Select one entry from the List,",
        "than press OK");
    if entry /= 0 then
        messageBox(values.read(entry),1,4);
    end;
    entries:=promptListn(values,
        "Select some entries from the List,",
        "than press OK");
    for i:=1 to entries.dim loop
        if entries.read(i) = 0 then
            exitLoop;
        else
            txt:=txt+values.read(entries.read(i));
            if i<entries.dim then
                txt:=txt+", ";
            end;
        end;
    next;
    messageBox(txt,1,4);
    --set the backgroundcolor of the frame
    self.~.backgroundColor:=
        openColorSelectBox(
            makeRGBValue(255,255,255));
end;
```

## 2.14.5 Output Functions

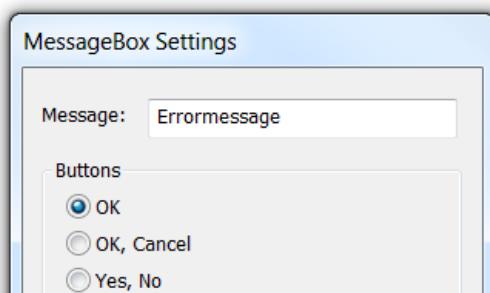
### 2.14.5.1 Messagebox

You can display a message box using:

```
Messagebox( "Message", integer buttons, integer icon);
```

Note: Plant Simulation shows a configuration dialog if you write messagebox( and press Ctrl + EmptySpace (Fig. 2.98);

```
is
do
  messagebox(
end;
```



**Fig. 2.98** MessageBox Configuration

The following values are used for buttons (Table 2.21):

**Table 2.21** MessageBox Button Values

Value	Description
1	OK
3	OK, Cancel
10	Repeat, Cancel
48	Yes, No
50	Yes, No, Cancel

For the icon, you can pass the following values (Table 2.22):

**Table 2.22** MessageBox Icon Values

Value	Description	Icon
0	no icon	
1	error	
2	question mark	
3	exclamation mark	
4	information	

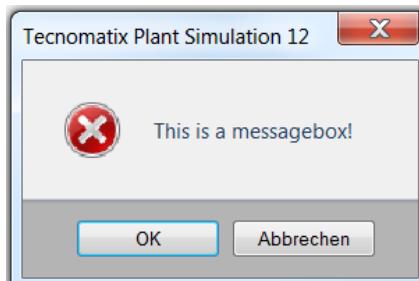
If the user clicks one of the buttons, the method messagebox returns one of the following values:

- OK—1
- Cancel—2
- Yes—16
- No—32

A Messagebox called with:

```
MessageBox("This is a messagebox!", 3, 1)
```

is shown as in Fig. 2.99.

**Fig. 2.99** MessageBox

Note: You can insert carriage returns and tabs into the text using the function Chr(ascii code). Enter Chr(9) for tabs and Chr(13) for carriage returns. Sample: "text1"+Chr(13)+"text2"

### 2.14.5.2 Infobox

In contrast to the messagebox, the infobox has no buttons. The syntax is:

```
Infobox(string message, boolean modal).
```

An infobox called by the command

```
infobox ("This is a message", false) looks like Fig. 2.100.
```



**Fig. 2.100** Infobox

An opened infobox can be closed again with:

```
infobox("", false);
```

If you pass the second parameter as true, then all dialogs are blocked in Plant Simulation. In this case, you must close the infobox with SimTalk (e.g. via the endSim method). The key combination Shift + Ctrl + Alt held down for five seconds closes an open infobox (as emergency variant).

### 2.14.5.3 Bell and Beep

Using the method `bell (<frequency>, <duration in ms>)` and `beep` you can play sounds over the sound system of the computer.

### 2.14.5.4 HTML Window

Plant Simulation provides a function to open a browser window and show a file in it. Syntax:

```
OpenHTMLWindow(<string location>, <string title>,
<integer x-Pos>, <integer y-Pos>, <integer width>,
<integer height>);
```

Example:

```
is  
do  
    OpenHTMLWindow("file:///c:/help.html", "Help",  
        200,200,300,200);  
end;
```

This opens a browser window (e.g., Internet Explorer). The HTML file will be loaded in this window. The HTML window can be closed with the command

```
CloseHTMLWindow(string title);
```

Example:

```
is  
do  
    closeHTMLWindow("Help");  
end;
```

# Chapter 3

## Modeling of Production Processes

It is perfectly possible to model a large number of different production processes using the basic equipment of Plant Simulation. In many cases, however, you must adjust the behavior of Plant Simulation objects in order to achieve a sufficiently accurate representation of the processes. The following chapter deals with the processing of one or more parts simultaneously, assembly and disassembly processes, and ensuring quality within the simulation.

### 3.1 Material Flow Library Elements

Movable and immovable material flow elements form the basic building blocks of a model. The movable material flow elements (vehicle, container, transport units) represent the physical or logical components that move through a model. These components are transported by active or passive material immobile flow elements through the simulation model (e.g. in the case of a part located on a conveyor, the passive part of the model is carried by the active element, the conveyor). Available active elements include, for example, SingleProc, ParallelProc, the assembly station, the dismantling station, the line, the rotary table, the transfer unit, and the buffer. These elements actively transport the movable material flow elements along the material flow connectors. Source and drain are used to create and destroy movable elements (BEs). They provide the model with boundaries. Passive material flow elements are the storage and the one- and two-lane tracks. These elements do not automatically transfer MUs. The block flow control, which itself cannot store MUs, is used to model the merge and distribution strategies.

#### 3.1.1 General Behavior of the Material Flow Elements

Active material flow elements can receive movable elements, store them for a certain time and then forward them automatically to the following element (successor). Passive material flow elements cannot move MUs automatically (e.g. an MU remains in the storage element until it is removed by a method). The passive element track can only be used effectively with the element transporter—i.e. this MU moves on the track at a certain speed.

### 3.1.1.1 Time Consumption

A material flow element accepts MUs if it has sufficient capacity and is not disturbed or paused. If one of the conditions is not met or the gateway is closed due to the cycle or recovery times, the building block rejects the MUs. The movable element is added to the end of a blocking list. If the building block can receive MUs again, the first MU in the blocking list will be moved and processed (first in, first out principle).

#### Example: Material Flow—Time Consumption

Set up a frame as shown in Fig. 3.1.

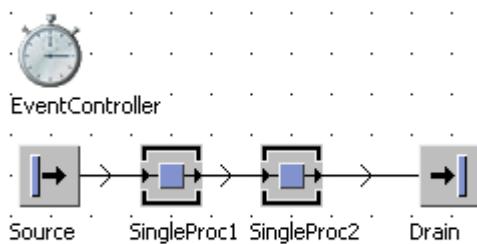


Fig. 3.1 Frame

Open the source with a double-click. The source generates MUs (by default entities). You can set here the distance between the individual entities. The default setting is zero minutes. Change the setting to one minute. In the next block (SingleProc1), set the processing time to two seconds. To do this, double-click the SingleProc1 in the frame. In the following menu, select the sheet Times. Enter 0:02 in the Processing time field. If the processing time of a following station is one minute, the MUs would "jam." Set the Processing time of SingleProc2 to two minutes. Let the simulation run for a while. Stop the simulation (click on the icon with the green triangle in the event manager), then open the SingleProc1. Select the sheet Statistics (Fig. 3.2).

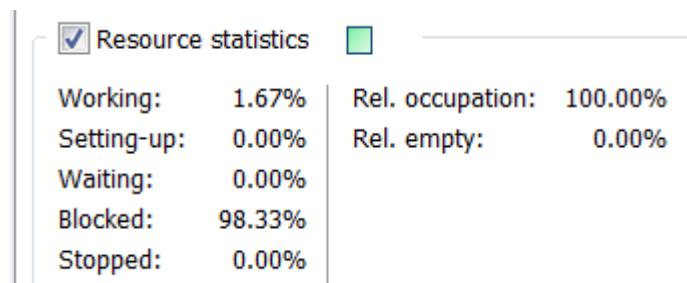


Fig. 3.2 Statistics Dialog

The SingleProc1 is blocked by its successor most of the time. It can pass the MU but the following station cannot receive the MU (because the station is still working). The elapsed time is referred to as blocking time.

### Entry Gates

There are two "gates" to enter a building block:

- The block is empty and not disturbed or paused.
- An integer multiple of the cycle time

The cycle time is used for synchronization. However, if one station has finished its operation earlier, the part waits until the cycle time is completely over. Only then is the part moved onto the next block.

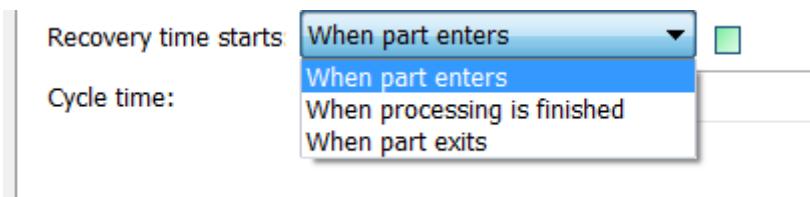
The duration of the work on a station consists of three parts:

### Set-up Time

The setup time is the time required to set up a basic block for processing a different type of MU. The type is determined by the name of an MU (MUs with the same name have the same type). You can also consider a setup, for example, to represent regular tool change after a certain number of parts.

### Recovery Time

At the entrance of a basic block is a gate that closes the entrance for a certain time. Starting with Version 10, you can assign this time to different events (Fig. 3.3).



**Fig. 3.3** Recovery Time

With the setting "If part enters," you can take into account times that are not processing times but connected to the single part on the station (for each processing). This allows one to model robots that need a certain amount of time for loading parts, along with clamping times, times for closing the doors, and more. The setting "When processing is finished" is useful if you need to represent times that elapse while the part is still on the machine and the processing has been completed. Such times could include the time required for the de-clamping of the part, for the opening of doors, etc. "When part exits" is used if the machine is

already empty, before the next occupancy. Often, machines need to be cleaned before the next operation can start or settings need to be changed. If this is true for all parts to be machined, then this is the right setting for it.

### Processing Time

The processing time determines how long an MU (possibly after a set up) remains on the station, before Plant Simulation tries to pass it to a successor.

#### 3.1.1.2 Capacity

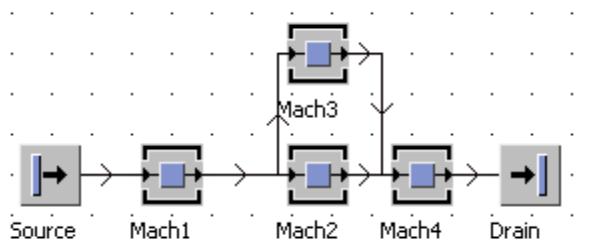
The capacity determines how many MUs can occupy the same station. If the capacity limit is reached, no more MUs are moved to the station. The dimension of the MUs (for now) is not considered (the MU is already located on the block if the first "tip" is moved to the block). For length-oriented blocks like track, turntable and line, the length of the MUs is of importance for calculating the capacity (attribute length).

#### 3.1.1.3 Blocking, Exit Behavior

MUs that want to enter into a full block will be rejected (and registered on a blocking list). If there is a branch in the network (several successors), the transfer request is made to the following blocks (in the default output behavior). The MU is then transferred to the next free block.

##### Example: Blocking

Create a frame according to Fig. 3.4.



**Fig. 3.4** Frame Example for Blocking

Processing time: Mach1 and Mach4 each 2:30 minutes; Mach2 and Mach3: five minutes; the source produces parts with an interval of 2:30 minutes. Block Mach2 in the sequence (set a checkmark on the box Failed, then click on Apply), and watch what happens.

You can control the behavior of the distribution of Mach1 on the tab Exit strategy. There are a variety of strategies (Fig. 3.5):

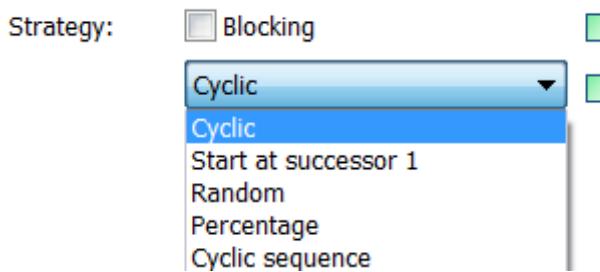


Fig. 3.5 Exit Strategy

### Blocking

If blocking is selected and an MU transfer request could not be finished, the transfer request will be entered in the blocking list of the block and will wait until the block is receptive again. If blocking is not selected, a transfer takes place when any follower can receive the MU. Furthermore, there are the following types of blocking (Fig. 3.6).

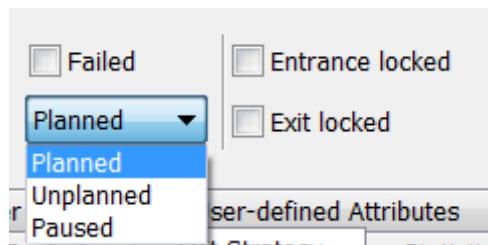


Fig. 3.6 Blocking Types

### Failed

The block cannot receive MUs; hence, already processed MUs can be forwarded. As long as the block is failed, the setup and processing time is stopped until the end of the failure (e.g. so that you can simulate what happens when a machine fails). Failed blocks are marked by a red LED.

### Paused

You can send a block into a pause condition (blue LED). Thereby, the processing time and setup time are interrupted, finished parts can be transferred, and the block cannot accommodate MUs. The same effect can be Unplanned (see shift calendar). With Unplanned, times outside the working hours are simulated. The Event Manager resets all failures and pauses if you restart the simulation.

### Entrance Locked

It is not possible to move the MU to the block. The MU is added to the blocking list and processed from there after releasing the lock.

### 3.1.1.4 Failures

To achieve a most realistic simulation, you must include some events that disrupt the normal flow of materials. You can position these events accurately or randomly. You can take into account times for retooling as well as maintenance times, accidents, machine failures, and more.

There are two possibilities for modeling failures:

- Using statistical distributions
- Using the mean time to repair (MTTR) and the availability

The dialogs of the material flow objects provide the tab Failures.

#### Example: Failure 1

A machine needs maintenance every 1,000 operating hours for duration of three hours. It requires these settings according to Fig. 3.7.

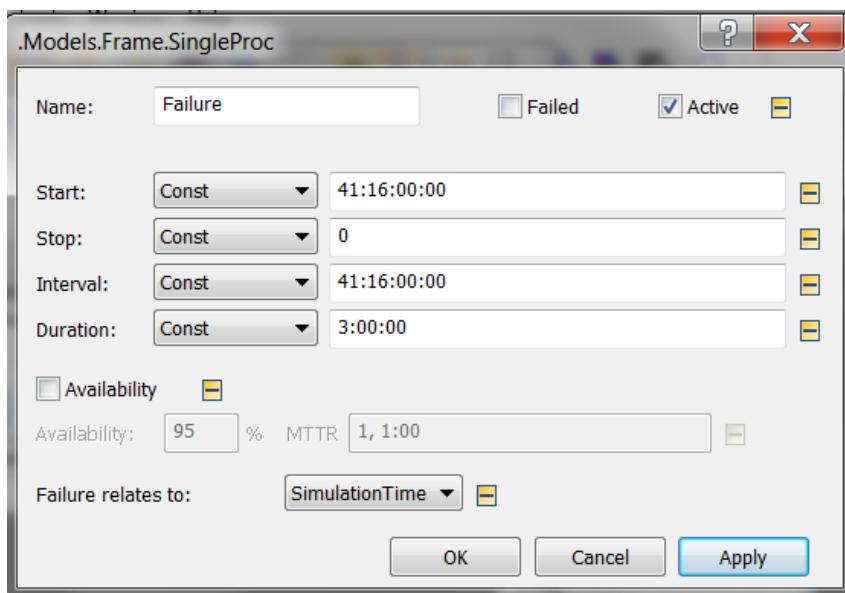


Fig. 3.7 Failure

Active: The check box turns all kinds of failure events on or off.

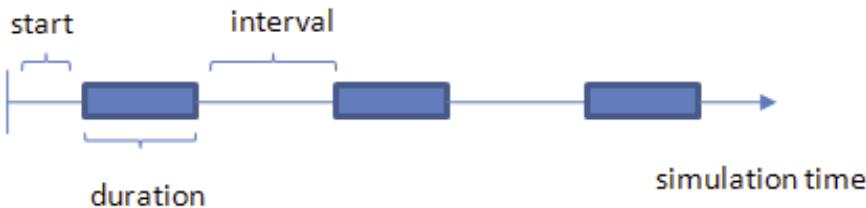
Start: With "Start," you can define the beginning of the failure. You can also select a statistical distribution.

Stop: End of failure

Interval: Enter an interval between the end of the last failure and the beginning of the next failure (trouble-free time). If the value of the selected interval is zero and

the value of the selected duration is greater than zero, then a single failure occurs. See also Fig. 3.8.

Duration: Duration of the failure (duration = zero; this means no failure).



**Fig. 3.8** Failure Settings

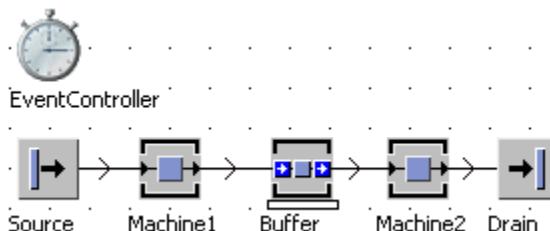
Failure relates to simulation time: For this setting, Plant Simulation consumes the time you have entered as the interval, independent of the state of the object (paused or operational).

Failure relates to processing time: For this setting, Plant Simulation consumes the time you have entered as the interval while the object is working (not paused, waiting, or unplanned).

Failure relates to operating time: For this setting, Plant Simulation consumes the time you have entered as the interval, if the object is not paused (working or waiting).

### Example: Failure 2

A machine (three-shift mode) needs maintenance lasting 1.5 hours after every 22.5 hours of processing time. The succeeding machine requires 30 minutes maintenance every 3.5 hours of processing time. Both machines have a processing time of two minutes. To ensure smooth material flow, a buffer is located between machine1 and machine2. How many places must the buffer have? Create a frame like in Fig. 3.9.



**Fig. 3.9** Frame Example: Failure 2

Let the simulation run for two days to observe the buffer. The line should not jam; the buffer should not be oversized either. You can evaluate the necessary buffer size using the tab Statistics of the object buffer.

### Availability (MTTR, MTBF)

You can specify the availability and average repair time for the processing stations (Fig. 3.10). The system then calculates a mean time between failures (MTBF). The duration of outages and failure-free times will be randomly distributed. You have to specify a random number stream (a random number stream is a series of random numbers). The availability will be calculated using the following formula:

$$\text{Availability} = \text{MTBF}/(\text{MTBF}+\text{MTTR}).$$

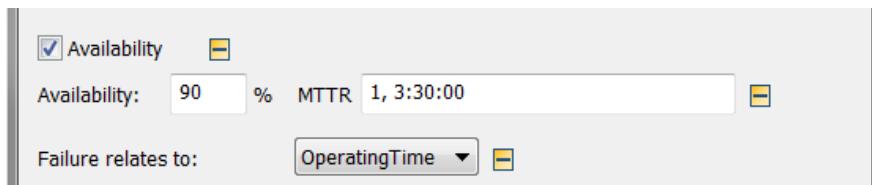


Fig. 3.10 Failure Availability

Enter a number between zero and 100 per cent for the availability. The availability is the probability that a machine is ready to use at any time. The availability is based on a combination of MTBF and MTTR. The duration of the failure as well as the distance between the failures are randomly distributed. Plant Simulation selects the Erlang distribution for the duration and the Negexp distribution for the interval.

Starting with Version 9 of Plant Simulation, you can create a series of failures for an object. In this way, you can more realistically model the failure behavior of machines and plants. You can, for example, set the maintenance intervals and tool changes of a machine in one dialog and without programming failures.

### Example: Multiple Failures

We will use a machine with the following failure behavior: For each five hours of processing time, a tool change takes place lasting 30 minutes. After every 1,000 hours of operating time, regular maintenance of two hours takes place, along with a five per cent loss (random) relative to the operating time and two hours MTTR. To consider these values in the simulation, you need to proceed as follows in Plant Simulation starting from Version 9. Click the tab Failures and then the button New. Enter the failure data into the dialog—e.g. tool change. All failures are displayed in a list. Double-clicking an item on the list allows you to edit the individual failures (Fig. 3.11).

Active	Name	Avail...	MT...	Mode	Start	...	Inter
<input checked="" type="checkbox"/>	Failure	95.00%	30:00	OperatingTime	0	0	
<input checked="" type="checkbox"/>	Tool_Change			ProcessingTime	0	0	5
<input checked="" type="checkbox"/>	Maintenace			OperatingTime	41:16:00:00	0	416:16

Fig. 3.11 Failure Profile

### 3.1.2 ShiftCalendar

Every material flow object that “deals with” entities has the following times:

- Planned (working within the shifts)
- Unplanned (times outside the shifts, e.g. weekend)
- Paused (pause within the shifts)

The ShiftCalendar sets these times using a TimeSequence. You can use one ShiftCalendar for the entire simulation, or, in extreme cases, create an individual ShiftCalendar for each machine.

#### Example: ShiftCalendar

You are to simulate a continuous process (coating), which has a workplace for preparation and another for follow-up jobs. A coating process takes eight hours (the facility is 75 m long), the preparing and follow-up job each take 2:30 minutes. The coating facility works 24 hours a day, seven days a week. The preparing and follow-up workplaces work according to the following shift system: The beginning of the first shift is Monday 6.00; the end of the last shift is Saturday 6:00. The morning shift starts at 6.00 and ends at 14.00 with breaks from 9:00 to 9:15 and from 12:00 to 12:30. The middle shift starts at 14.00 and ends at 22.00, with breaks from 17.00 to 17.15 and from 20:00 to 20:30. The night shift begins at 22.00 and continues until 6.00, with breaks analogous to the middle shift. What is the maximum output? Create a frame according to Fig. 3.12:

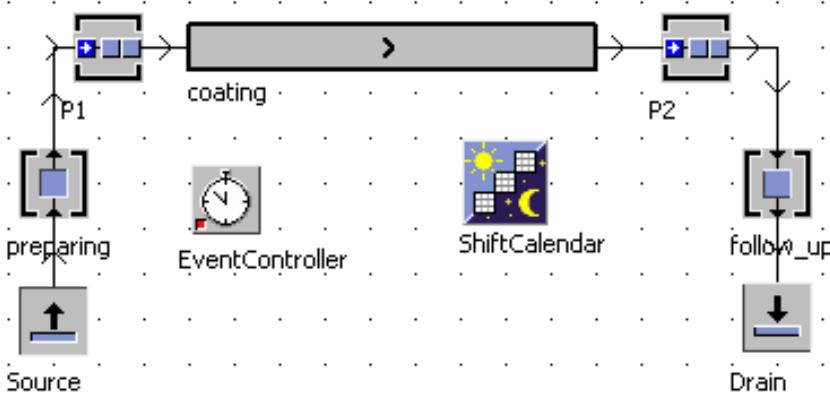


Fig. 3.12 Frame ShiftCalendar

Settings: Length of the entities: 0.2 meters; P1, P2 processing time: 0 seconds; capacity: 10,000 parts each. Insert a ShiftCalendar object into the frame. First switch

off inheritance on the tab Shift times (click on the green icon on the right side + Apply). Then enter the shift times into the table (Fig. 3.13).

Shift Times		Calendar		Resources		User-defined Attributes						
	Shift	Fro...	To	M..	Tu	W.	Th	Fr	S	S	Pauses	
1	Shift-1	6:00	14:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	9:00-9:15; 12:00-12:30					
2	Shift-2	14:00	22:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	17:00-17:15; 20:00-2...					
3	Shift-3	22:00	6:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	01:00-01:15; 04:00-0...					

**Fig. 3.13** ShiftCalendar Settings

Assign the ShiftCalendar to the objects on the tab Controls—Shift calendar. The tab Resources of the ShiftCalendar provides an overview of the stations that use the ShiftCalendar.

The following methods and attributes are provided by SimTalk for working with the ShiftCalendar (Table 3.1):

**Table 3.1** ShiftCalendar Attributes and Methods

Method/Attribute	Description
<path>.getCurrShift	Returns the name of the current shift (as string), if the time of the query is not working time and returns an empty string
<path>.calculateWorkingDuration( <datetime1>, <datetime2>)	Calculates the available working time between two points in time (datetime1, datetime2)
<path>.schedule(<datetime>, <time>, <string>)	Calculates the duration of a job starting from a date (datetime) and a specific duration (time). The last parameter specifies the scheduling direction ("forward" or "backward")
<path>.shiftPlan	Copies the contents of a table in the shift schedule; you need to format the table (see help)
<path>.shiftCalendarObject	Returns the assigned shift calendar for an object

Unfortunately, the ShiftCalendar does not provide an observable attribute for detecting a shift change. A detour would be to call the method `getCurrShift` periodically to save the current shift in a global variable. These can then be monitored with an observer to respond to the change of shift. The call can also be implemented in an entrance control (e.g. drain).

### Example: ShiftCalendar—Shift Change

You want to save the output per shift in a table. The production works two shifts. You can use the default setting for the ShiftCalendar. Create a simple network according to Fig. 3.14 Frame Shift ChangeFig. 3.14.

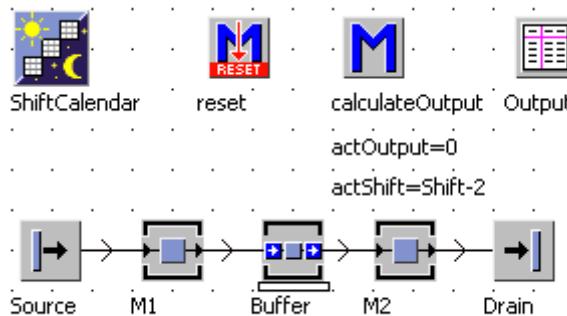


Fig. 3.14 Frame Shift Change

Settings: source interval: two minutes; machine1 processing time: one minute, 50 per cent availability, one minute MTTR; Machine2 processing time: one minute, 50 per cent availability, three hours MTTR; buffer: capacity: 1,000 parts; shift calendar: Default Settings. Format the table output according to Fig. 3.15.

	date 1	string 2	integer 3
string	Date	Shift	Output
1			

Fig. 3.15 Format of Table Output

The global variable `actOutput` has the data type integer and an initial value of zero. The global variable `actShift` has the data type string and an empty string as the starting value. The reset method must clear the output table before the start of a new simulation. Reset method:

```
is
do
  output.delete;
end;
```

The method `calculateOutput` is the entrance control of the sink. It counts the parts received within the shift and writes at shift change the date, the shift and the number of parts in the output table. One problem with this approach is that the number of parts of the second shift of each day is stored only if the method is triggered in the first shift of the following day. Therefore, the date must be corrected for each second shift. Method `calculateOutput`:

```

is
  dat:datetime;
do
  dat:=EventController.AbsSimTime;
  if shiftCalendar.getCurShift = actShift then
    actOutput:=actOutput+1;
  else
    -- shift change
    if actOutput > 0 then
      if actShift ="Shift-2" then
        dat:=output[1,output.yDim];
      end;
      -- write row
      output.writeRow(1,
        output.yDim+1, dat, actShift, actOutput);
    end;
    -- new Shift
    actShift:=ShiftCalendar.getCurShift;
    -- start new count
    actOutput:=1;
  end;
end;

```

The `calculateWorkingDuration` method has a large amount of applications. You can use this method

- to calculate a cycle time based on a weekly output
- to empty a line before the end of a shift
- or avoid an occupation of a machine immediately prior to starting a break

#### Example: Weekly Output—Cycle Time

Calculate within the simulation the cycle time based on a weekly output and a given working time regime. To do this, create a simple frame like the one shown in Fig. 3.16.

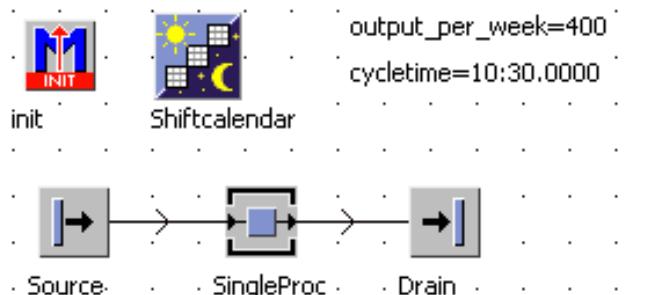


Fig. 3.16 Frame Example

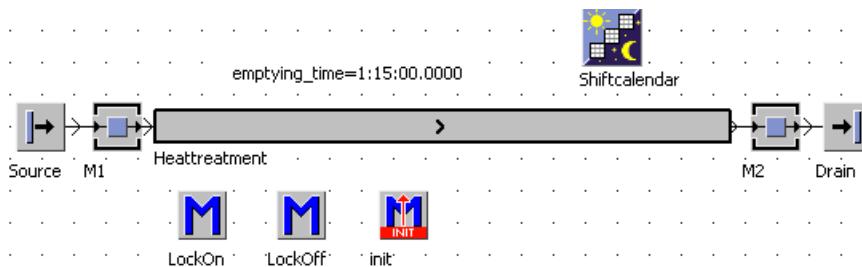
Set an availability of 95 per cent and an MTTR of one hour in the single station. You can use the calendar in the default setting. For calculating weekly working time, take a week from the past as a reference. The cycle time you can calculate using the weekly output. In the init method, this would appear as follows:

```
is
  worktime:time;
do
  --calculate worktime for a reference week
  worktime:=Shiftcalendar.calculateWorkingDuration(
    str_to_dateTime("07.01.2013 00:00"),
    str_to_dateTime("14.01.2013 00:00"));
  cycletime:=worktime/output_per_week;
  --take availability into account
  SingleProc.procTime:=cycletime*
    (SingleProc.failures.failure.availability/100);
end;
```

Set experiment duration of one week. Check the result. Analogously, you could proceed if you need to calculate the cycle time based on a daily output.

### Example: Emptying a Line before the End of a Shift

Often, parts of the plant must be run empty before the end of the shift (start time of unplanned) to prevent damage of the parts. For this purpose, the filling of the system components must be finished in a time interval (empty drive) before the end of the shift. Within the simulation, you can calculate the remaining available time in the shift and compare it with the empty drive time. When the remaining time is less than the empty drive time, lock the input of the relevant part of the plant. Set up a frame according to Fig. 3.17.



**Fig. 3.17** Frame Example

Ensure the following settings: M1 and M2 processing time: 10 minutes; heat treatment lead time: one hour. Assign the ShiftCalendar to M1, M2 and heat treatment. Insert one shift (without breaks) in the ShiftCalendar for Monday to Friday from 06:00 to 22:00. The method LockOn is the exit control of M1.

LockOff is the observer method of heat treatment (assigned to the property Unplanned). The method LockOn has the following content:

```

is
do
  --if remaining working time is less than
  --emptying_time then
  --not move --> lock entrance of heat treatment
  if shiftcalendar.calculateWorkingDuration(
    eventcontroller.absSimTime,
    (eventcontroller.absSimTime+emptying_time)) <
    emptying_time then
    heattreatment.entranceLocked:=true;
    --wait for start of the next shift
    waituntil heattreatment.entranceLocked=false
    prio 1;
    @.move;
  else
    @.move;
  end;
end;

```

You can identify the start of the shift using the property Unplanned. You can use an observer (object heat treatment) to call a method after the start of the shift (unplanned changes from true to false). The necessary setting is shown in Fig. 3.18.

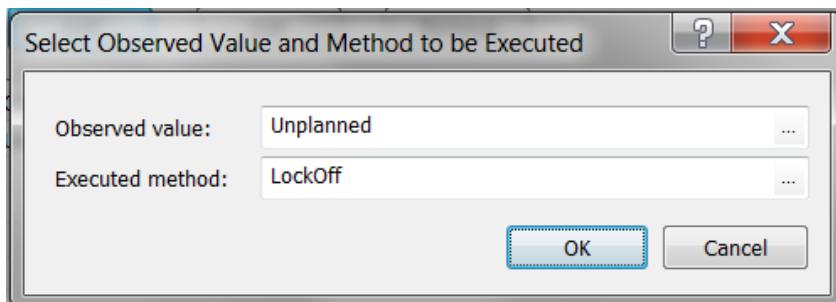


Fig. 3.18 Observer Setting

The method LockOff sets the attribute EntranceLocked of the heat treatment object to false, so that new MUs can enter.

```

(Attribute: string; oldValue: any)
is
do
  if heattreatment.unplanned=false then
    heattreatment.entranceLocked:=false;
  end;
end;

```

EntranceLocked is not automatically set to false when you start a new simulation run; you will have to do it using the init method:

```
is
do
  heatTreatment.entranceLocked:=false;
end;
```

### Example: Remaining Time until Break

You can also use calculate WorkingDuration to prevent an occupation of machines before breaks/the end of the shift, if the time period is not sufficient to process the MUs completely until the start of the break/the end of the shift. This mode is important when no parts are located inside the machine during the break. To ensure this, verify that the processing time of the station without "interruption" is available (Time = working time). Set up a simple frame according to Fig. 3.19.

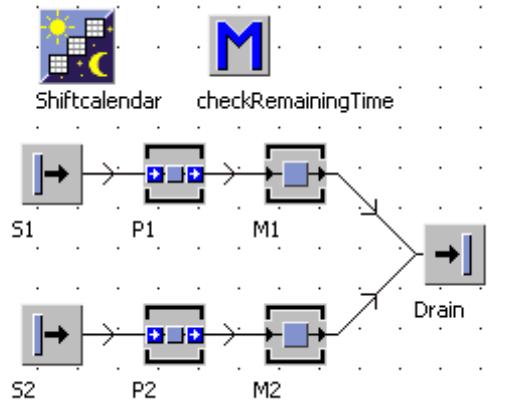


Fig. 3.19 Frame Example

Introduce the following settings in the model: M1 and M2: processing time: 5:45; set the ShiftCalendar for the stations. The method checkRemainingTime is the exit control (front) of the buffer (P1 and P2). The ShiftCalendar and all other objects can use the default settings. The method checkRemainingTime compares the time period available for processing with the processing time. If the time is too short for processing, the method waits twice: first, until the break begins, and then until the break ends. Method checkRemainingTime:

```
is
  s:object;
  remainingTime:time;
do
  s:=?.succ;
  --calculate the remainingTime within the procTime
```

```

if s.shiftCalendarObject /= void then
  waituntil s.empty and s.operational prio 1;
  remainingTime:=
    s.shiftCalendarObject.calculateWorkingDuration
    (eventController.absSimTime,
     eventController.absSimTime+s.procTime);
  if remainingTime < s.procTime then
    --wait for pause
    waituntil s.operational=false prio 1;
    --wait for start workingtime
    waituntil s.operational prio 1;
  end;
end;
@.move;
end;

```

## 3.2 SimTalk Attributes and Methods of the Material Flow Elements

All attributes that you can set in the dialogues of the material flow objects can also be set by SimTalk. This can be used, for example, to load the basic settings of the objects from a central location (e.g. a table). The SimTalk attributes are similar to the labels on the dialogs. Important SimTalk attributes are:

- ProcTime (processing time)
- RecoveryTime (recovery time)
- CycleTime
- Capacity (if defined)
- MTTR
- MTTR.stream
- Availability

### 3.2.1 States of the Material Flow Elements

From Version 9, you can create for each object a series of failures. You get access to the single failure on a failure listing (failures). The individual failures are addressed by name. If your failure is, e.g. failure1, the failure is made accessible by failures.failure1. Each failure has the properties of availability, MTTR and MTTR.stream.

### Example: Basic Settings

The data (processing time, recovery time, and one failure) of three objects will be read from a table when initializing the frame. Create a frame according to Fig. 3.20:

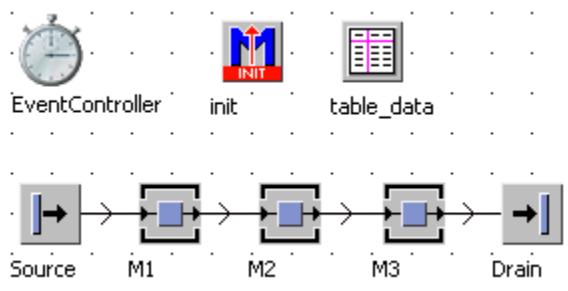


Fig. 3.20 Frame Example

From Version 9, you must first create a failure for M1, M2, and M3. Name the failure in all three objects “failure1.” Format the table according to Fig. 3.21 (take note of the data types) and enter the data (see List Editor in the chapter on Information flow objects):

	object <sub>1</sub>	time <sub>2</sub>	time <sub>3</sub>	real <sub>4</sub>	time <sub>5</sub>	integer <sub>6</sub>
string	Machine	Processing_time	Recovery_time	Availability	MTTR	Stream
1	M1	2:00.0000	10.0000	95.00	1:00:00.0000	9
2	M2	1:30.0000	20.0000	80.00	30:00.0000	10
3	M3	1:00.0000	10.0000	60.00	45:00.0000	11

Fig. 3.21 Content of table\_data

M1, M2, and M3 must have from Version 9 a failure “Failure” (Fig. 3.22).

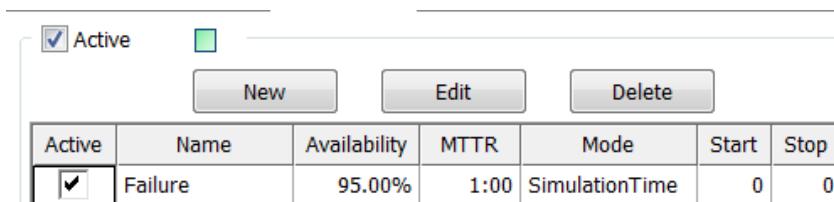


Fig. 3.22 Failure

The init method should appear as follows (see also the chapter Table):

```

is
  i:integer;
  machine:object;
do
  for i := 1 to table_data.yDim loop
    machine:=table_data["Machine",i];
    machine.procTime:=
      table_data["processing_time",i];
    machine.recoveryTime:=
      table_data["Recovery_time",i];
    /* version up to 8.2
    machine.availability:=
      table_data["Availability",i];
    machine.MTTR:=table_data["MTTR",i];
    machine.MTTR.stream:=table_data["Stream",i];
    */
    -- from version 9
    machine.failures.failure1.availability:=
      table_data["Availability",i];
    machine.failures.failure1.MTTR:=
      table_data["MTTR",i];
    machine.failures.failure1.MTTR.stream:=
      table_data["Stream",i];
  next;
end;

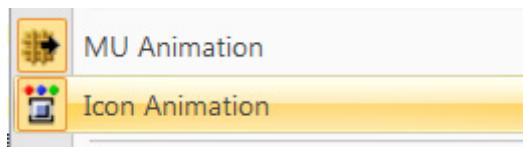
```

Starting from Version 11, the setting of the stream is no longer necessary.

**Table 3.2** Methods and Attributes for the States of the Material Flow Elements

Method/ Attribute	Description
<path>.operational	The method returns TRUE as a result if the element is neither failed nor paused; otherwise, the result is FALSE.
<path>.failed	This property sets and reads the state failed.
<path>.pause	This sets and reads the state pause
<path>.empty	The method empty returns TRUE if no MU is wholly or partly located on an element. You can also address certain places, for example, on a parallel station (with x-y coordinates).
<path>[int1,int2].empty	

Plant Simulation normally signals the operating status of the objects with different colored LEDs. This function you can activate/deactivate in Plant Simulation with View—Icon Animation (Fig. 3.23).



**Fig. 3.23** View—Icon Animation

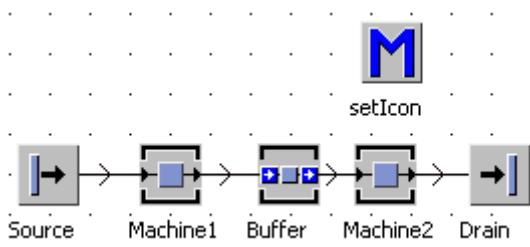
If you need to work with very small icons, then Plant Simulation reduces the size of the LEDs beyond recognition. The best option then is to change the symbol entirely if the state of the object changes. There are two approaches in Plant Simulation:

- You define icons for the different states and Plant Simulation automatically changes the icons when the state changes (only standard conditions, only a few states).
- You change the icons using SimTalk (flexible).

The following example provides the approaches to do this.

### Example: Icon Animation

To display the status of material flow components, exchange the icons of the objects. First, create a set of icons for the various states in the class of the SingleProc in the class library (Context menu—Edit icons—Icon—New). Create the symbols in different colors. Use blue for pauses (symbol number 2, the name "pause") and red for failed (symbol number 3, the name "failed"), and green for working (symbol number 4, the name "working"). Other states are observable—e.g. set up (setup), ready (empty, operational) and locked entrance or recovery time (no\_entry). Beware that the animation points of all the symbols are at the same position. Create a frame according to Fig. 3.24.



**Fig. 3.24** Frame Example

Settings: source interval: two minutes; machine1 and machine2: one minute processing time, 50 per cent availability, 30 minutes MTTR; buffer: capacity 100 units, no processing time.

### Variant 1: State Icons

To display the states, you can choose between LED and icon animation. In the setting icon animation, the state of the object is visualized by changing the entire symbol. The icon animation can represent only one state, while LED can also display combinations of states. You can map only the above mentioned states.

To activate the icon animation, proceed as follows: Open the Symbol Editor (context menu—edit symbols) for the single station in the class library. Select in the Symbol Editor Object—State Icons/LEDs—Use State Icons (Fig. 3.25).

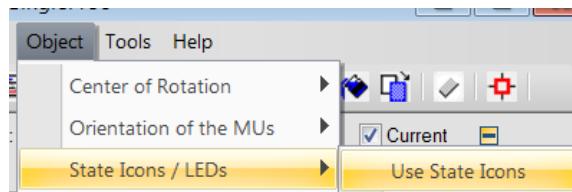


Fig. 3.25 Setting Icon Editor

Now, when a failure occurs, Plant Simulation shows the icon labeled "failed" and changes to "operational" or "working" if the failure is over. More flexible is the realization with SimTalk.

### Variant 2: Observer and SimTalk

After changing the properties, the failed or paused state of Machine1 should be called the method setIcon. This works best with an observer. Open the dialog of machine1 and select Tools—Select Observers (Fig. 3.26).

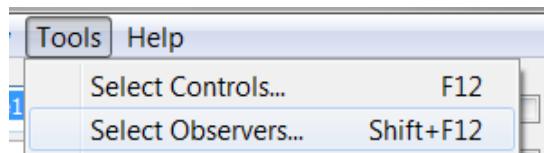


Fig. 3.26 Observer

Click Add and select as the observed value Failed, as shown in Fig. 3.27.

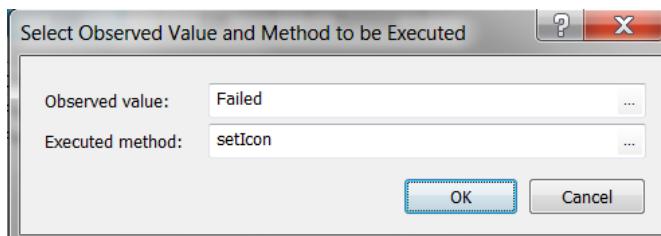


Fig. 3.27 Observer Setting

Analogously, you proceed for the attribute pause and Machine2. In the method setIcon, load the correct icon for the state of the machine.

- Neither failed nor paused: Icon 1
- Paused: Icon 2
- Failed: Icon 3

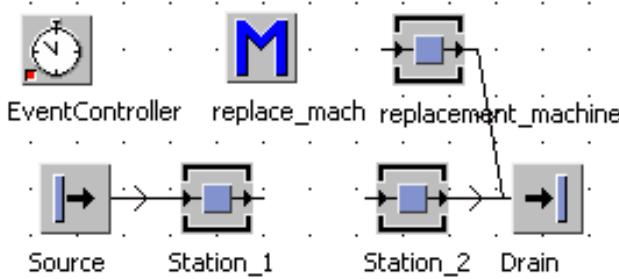
A “?” can be used to request the calling machine. In this example, the method should appear as follows:

```
(attribute: string; oldValue: any)
is
do
  if ?.pause then
    ?.CurrIconNo:=2;
  elseif ?.failed then
    ?.CurrIconNo:=3;
  else
    ?.CurrIconNo:=1;
  end;
end;
```

An important attribute is operational. You need this attribute if you want to query whether an object is able to receive MUs. If you try to move (using SimTalk) an MU on a failed or paused object, then the method move returns false and the moving of the MU fails.

### Example: Replacement Machine

Station\_1 usually delivers parts to Station\_2. If Station\_2 is failed, Station\_1 should deliver to replacement\_machine. Create a frame according to Fig. 3.28.



**Fig. 3.28** Example Frame

Method replace\_mach—exit control front Station\_1

```

is
do
  if Station_2.operational=false then
    @.move(replacement_machine);
  else
    @.move(Station_2);
  end;
end;

```

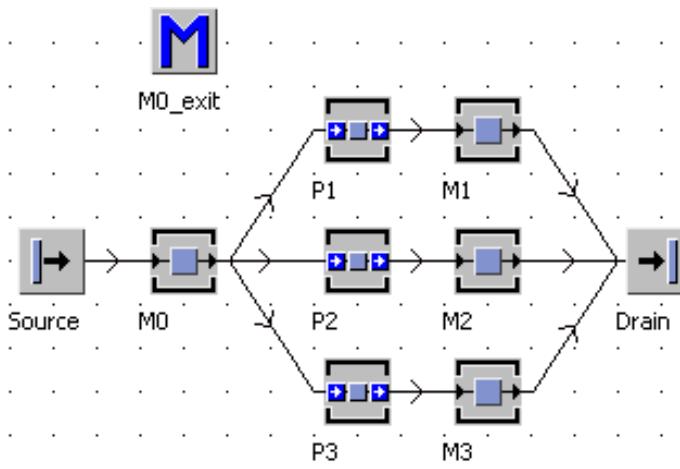
Activate failures during the simulation for Station\_2. The parts are redirected to the replacement machine.

### Remaining Processing Time

To determine the time necessary to complete the processing time, you can use the attribute RemainingProcTime of the MUs. For example, you can use this attribute to control the occupation of machines.

#### *Example: Remaining Processing Time*

In a production, the loading of the machine should take into account the remaining processing time of the single machines. Create a simple frame according to Fig. 3.29.



**Fig. 3.29** Example Frame

Create the following settings: source: interval one minute; M0: processing time one minute, exit control: front- M0\_exit; P1, P2, P3 each at one place, no processing time; M1, M2 and M3: processing time uniformly distributed between one and five minutes across different random number streams (until Version 11).

If a buffer and the corresponding SingleProc are empty, then the MU will be moved to this buffer. If the buffer capacity is available and the SingleProc occupied, then the lowest remaining processing time of the SingleProc stations decides the destination of the MU. This logic could be implemented as follows:

```

is
  i:integer;
  minSucc:integer;
  remainingTime:integer;
do
  --one of the buffers has to be empty
  waituntil P1.empty or P2.empty or P3.empty prio 1;
  --move directly if the buffer and the
  --singleproc are empty
  --succ(x) grant access to all successors
  for i:=1 to ?.numSucc loop
    if ?.succ(i).empty and ?.succ(i).succ.empty then
      @.move(??.succ(i));
      return;
    end;
  next;
  --not yet moved
  remainingTime:=1000000; -- error value
  --calculate remainig procTime and min. value
  for i:=1 to ?.numSucc loop
    if ?.succ(i).empty and
      ?.succ(i).succ.occupied then
      if ?.succ(i).succ.cont.RemainingProcTime <
        remainingTime then
        remainingTime:=
          ?.succ(i).succ.cont.RemainingProcTime;
        --save succ-No
        minSucc:=i;
    end;
  end;
  next;
  --move to the successor with the lowest procTime
  @.move(??.succ(minSucc));
end;

```

### 3.2.2 Setup

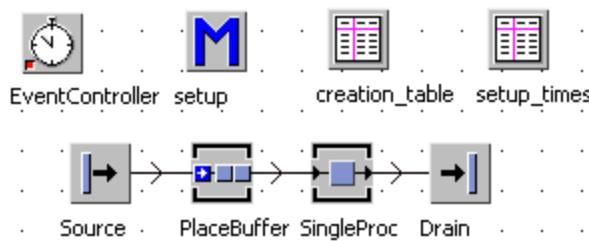
For the setup, the following attributes and methods are available (Table 3.3):

**Table 3.3** SimTalk Attributes and Method for Setup (Selection)

Attribute/Method	Description
<code>&lt;path&gt;.setup</code>	Determines whether an element is currently in setup mode (true if the element currently is in mode setup).
<code>&lt;path&gt;.setupFor(object)</code>	Triggers a setup for a particular MU class. The time required depends on the value of the setup time of the element. You need to take over the MU class.
<code>&lt;path&gt;.isSetupFor</code>	Returns the MU name for which the block is set up. If the block is not prepared for a MU, an empty string is returned.
<code>&lt;path&gt;.setupTime</code>	Sets/reads the setup time of the block.
<code>&lt;path&gt;.automaticSetup</code>	Indicates whether the setup of the block is automatically triggered when a new MU (an MU with another name) reaches the block.
<code>&lt;path&gt;.remainingSetupTime</code>	Returns the time until the end of the setup process.

#### Example: Setup at Lot Change

A milling center successively processes different production orders. The parts call for different setup times. Prior to the part moving onto the machine, the setup must be set anew if necessary. The necessary information is to be centrally stored in a TableFile. Create a frame like in Fig. 3.30:



**Fig. 3.30** Example Frame

Settings: source interval: 1:30 minutes, blocking; create three entities p1, p2, p3. The source creates the parts in a cycle. Use the TableFile creation\_table for the distribution of the parts. Type the lot sizes into the TableFile creation\_table according to Fig. 3.31.

	object 1	integer 2	string 3	table 4
string	MU	Number	Name	Attrib
1	.MUs.P1	5		
2	.MUs.P2	10		
3	.MUs.P3	5		
4				

**Fig. 3.31** Creation Table

PlaceBuffer: capacity 100 parts, no processing time; the TableFile setup\_time contains the information in Fig. 3.32.

	string 0	time 1	time 2
string		setup_time	dismantling_time
1	P1	10:00.0000	5:00.0000
2	P2	13:00.0000	3:00.0000
3	P3	20:00.0000	5:00.0000

**Fig. 3.32** Setup Times

The setup time consists of the dismantling time of the old part (already located on the machine) and the setup time for the new part. Setting up from P1 to P2 might take, for example, 5+13 minutes = 18 minutes. The setup time must be assigned to the workstation before the machine setup (automatically, every time an MU with a different name arrives). For this reason, we have to make the following considerations: It has to be checked for which part (MU) the machine is equipped. If the machine is set up for the same part, then no action is required; if the machine is set up for another part, the setup time is set anew and the machine starts setting up after moving the part to the machine. Program the method set-up (as the exit control front for the PlaceBuffer):

```

is
  former : string;
do
  -- read the set-up time from the table
  -- set the attribute setupTime
  former:=singleProc.isSetUpFor;
  if former="" then
    --the first part only set-up
    singleProc.setupTime:=
    setup_times["setup_time", @.name];
  else
    -- former part dismantling_time

```

```
-- recent part setup_time
singleProc.setUpTime:=setup_times["setup_time",
@.name] + setup_times["dismantling_time",former];
end;
@.move;
end;
```

Note: For the distribution of setup times, depending on the actual part and the new part, you can define the setup times in a matrix.

### 3.2.3 *Finished Messages*

#### 3.2.3.1 *Finished Messages Using resWorking*

A number of functions provide an indication of the state of the material flow objects (Table 3.4).

**Table 3.4** States of the Material Flow Objects

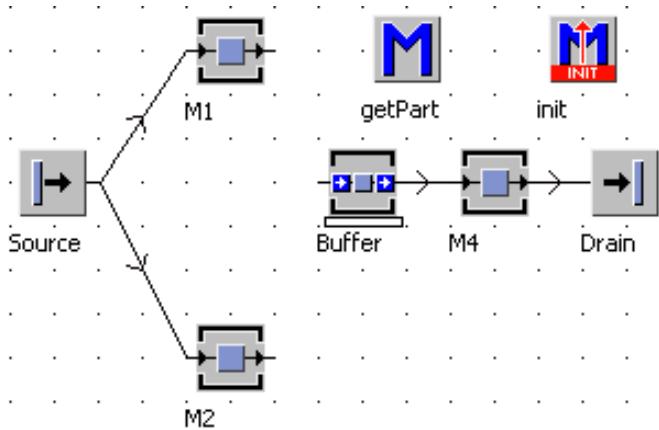
Attribute/Method	Description
<path>.resWorking	Returns true if the station is occupied and the processing time is not over (state working).
<path>.resBlocked	Returns true, if the station is blocked
<path>.resWaiting	Returns true, if the station is waiting
<path>.setup	Returns true, if the station is in setup mode

Since these four methods are observable, you can use them within `waituntil` statements or observers. By checking whether a station is occupied and `resWorking = false`, you can see that the processing is complete.

#### **Example: Use of resWorking**

Create a frame according to Fig. 3.33.

Create the following settings: source interval: 1:50; M1: processing time 4:00, 95 per cent availability, MTTR 15 minutes; M2: processing time 2:00, 98 per cent availability, MTTR 15 minutes; buffer: capacity 1; M4: processing time 1:50, 100 per cent availability. The method `getPart` is the output control (rear) of the buffer. The `init` method calls the `getPart` method once:



**Fig. 3.33** Example Frame

```
is
do
  getPart;
end;
```

In the simulation, unload the machine M2 with a higher priority than M1. For this, you must wait until one of the two machines is ready. Thereafter, unload the machine M2 first (method `getPart`):

```
is
do
  --move parts first from M2, then from M1
  waituntil (M1.occupied and M1.resWorking=false) or
    (M2.occupied and M2.resWorking=false) prio 1;
  if (M2.occupied and M2.resWorking=false) then
    M2.cont.move(buffer);
  else
    M1.cont.move(buffer);
  end;
end;
```

### 3.2.3.2 Create Your Own Finished Messages

The method `ready` returns TRUE if the object is occupied and an MU is ready to exit.

Syntax: `<Path>.ready;`

Unfortunately, `ready` is not observable. If you need an observable attribute that could be used to trigger an action once the machine has finished processing and the part is ready for exit (e.g. to request a transporter), you can employ a user-defined attribute. Insert into the class of the object (e.g., `SingleProc`) a user-defined attribute

(processingReady, data type Boolean, value false). Your user-defined attribute (data type Boolean) must be observable. It must be set at the entrance of an MU to false and then trigger the exit sensor to true. The method could appear as follows.

Entrance control:

```
is
do
?.processingReady:=false;
end;
```

Exit control:

```
is
do
?.processingReady:=true;
end;
```

You can monitor this attribute within a waituntil statement or by an observer.

### Example: Ready Message of a Machine

You want to simulate a worker. The worker loads and unloads a machine. He stands in front of the machine, removes parts from a conveyor belt, places them in the machine and waits until the machine has finished processing. Then, he removes the parts and places them after a brief inspection on a second conveyor belt. To simulate the worker in this example, use a simple SingleProc. Create a frame according to Fig. 3.34.

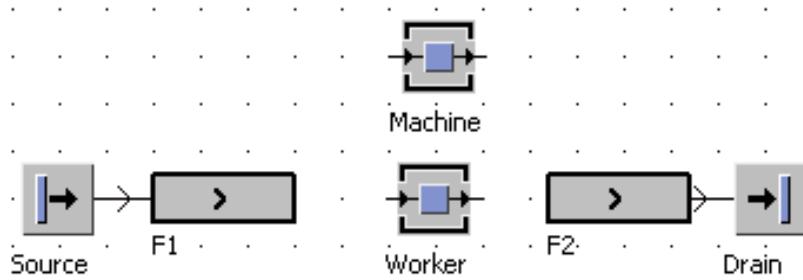


Fig. 3.34 Example Frame

Prepare the following settings: The source produces one entity every two minutes. F1 and F2 are conveyors with a length of 3 m and a speed of 1 m/s. The worker has a processing time of 20 seconds; the machine has a processing time of 1:20. Insert a user-defined attribute into the entity in the class library: OP, data type integer, initial value zero. Introduce in F1, worker and machine a user-defined attribute exitControl (data type method). Assign this method to the output control of these objects. Create in the machine, in addition, a user-defined attribute

machine\_ready (Boolean). In the output control of the conveyor F1, you control the repositioning of the parts on the worker. A new MU is moved when the worker is ready and neither workers nor the machine has loaded a part. Therefore, the output control of F1 would be as follows:

```
is
do
    waituntil machine.empty and worker.empty and
        Worker.operational prio 1;
    @.move(Worker);
end;
```

The worker "loads" the part twice. He loads the part once in order to load the machine, and later for a brief inspection (processing time) after unloading the machine. In such cases, you need one attribute to indicate the current state of the part (op). This attribute is changed after processing on the machine. In this example, an op = zero, means that the part must be processed on the machine. op = one, means that the part has to be moved to the conveyor F2. The output control of the worker might appear as follows:

```
is
do
    if @.op = 0 then
        -- to the Machine
        @.move(machine);
        -- waituntil ready, then unload the machine
        waituntil machine.machine_ready prio 1;
        machine.cont.move(self.~);
        -- reset the state of the machine
        machine.machine_ready:=false;
    else
        @.move(F2);
    end;
end;
```

The machine must now produce a ready message when the part is finished. For this, you can use the exit control. If the part is finished, Plant Simulation tries to transfer the part to the successor. Even if the object is not connected to any other element, the exit control is triggered. In the exit control, set the global variable machine\_ready to true and increase the OP. Therefore, the output control of the machine has the following content:

```
is
do
    @.op:=@.op+1;
    self.~.machine_ready:=true;
end;
```

### 3.2.4 *Content of the Material Flow Objects*

There are a number of attributes and methods that provide information about the content or allow access to the contents of an object (Table 3.5).

**Table 3.5** Attributes and Methods for Accessing the Content of an Object

Attribute/Method	Description
<path>.occupied	The method occupied returns TRUE if at least one MU is wholly or partially located on the object (otherwise, it returns FALSE). If the object has several places, each place can be queried individually (xy coordinates).
<path>[x,y].occupied	
<path>.full	The method “full” returns TRUE if the capacity of the object is exhausted. For a place-oriented object, the method “full” returns TRUE if any of its places are occupied.
<path>.numMu	The method numMU returns the number of MUs booked on the object.
<path>.capacity	The method “capacity” returns the number of places of an object.
<path>. deleteMovable	The method deleteMovable destroys all MUs that are booked on the object.
<path>.cont	Cont returns the MU, which is booked on the object. If there is no MU booked on the object, then the return value is VOID.
<path>.muPart(<integer>)	muPart returns the MUs that completely or partially are located on the object (important for length-oriented objects)
<path>.mu(<integer>)	The method MU accesses all MUs whose booking points are located on the object. If no argument is given, then the first MU is returned. If the argument is greater than the number of booked MUs, then VOID is returned.

## 3.3 Mobile Units (MUs)

MUs represent the materials that flow from object to object within the frame. After creating new MUs, they move through the model and remain at the end until they are destroyed.

### 3.3.1 *Standard Methods of Mobile Units*

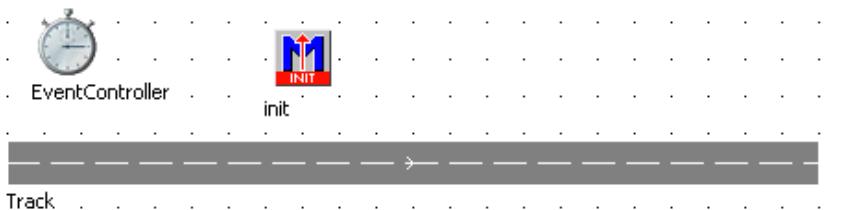
SimTalk provides a large number of attributes and methods for dealing with MUs. The most important applications are creating, destroying and moving MUs. In addition, you can use MUs as an information carrier through user-defined attributes and setting their values (Table 3.6).

### Create MUs

The method `<MU_path>.create(<object>[, length])` creates an instance of MU on `<object>`; MU path is the path to the MU (in the class library). Optionally, you can specify a length on a length-oriented object (e.g. track, line). If you do not specify a length, then the MU is generated at the end of the object (ready to exit).

### Example: Create MUs

At 200 m, 500 m, and 700 m, a transporter is to be created on a track which is 1,000 m long. Create a frame like in Fig. 3.35:



**Fig. 3.35** Example Frame

Note: If you want to create a track that is 1,000 m long by dragging, then you first have to change the scaling in the frame window. The default setting is a grid of 20 x 20 pixels and a scale of 1 m per grid point. Therefore, you have to change the scale so that it shows 50 m per grid point (Frame window, Tools—Scaling Factor). Another possibility is the following: On the tab Curve (track), turn off the option Transfer Length. Then scale a length of 1,000 m on the tab Attributes. The track is no longer shown to scale.

The init method is to first delete all MUs and then create the three transporters:

```
is
do
  deleteMovable;
  .MUs.Transporter.create(track,200);
  .MUs.Transporter.create(track,500);
  .MUs.Transporter.create(track,700);
end;
```

After creating the MU on a place-oriented object, it can exit immediately. On a buffer, MUs are created on the first free place, if no place has been specified. On a length-oriented object, the MU is created as close as possible to the exit if no position was specified. Creation will fail if the capacity of the object is exhausted and the length-oriented object is shorter than the MU to be created (Return value: VOID).

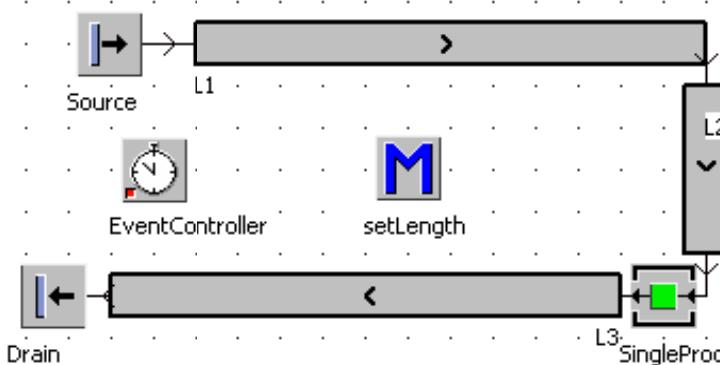
**Table 3.6** MU-related Attributes and Methods

Attribute/Method	Description
<MU-path>.delete	This method destroys the specified MUs. It does not delete MUs in the class library.
<MU-path>.move < MU-path>.move(<target>) <MU-path>.move(<index>)	Move the front of the transferred MU. If no argument is specified, then it will be transferred to each successor alternately. If it should be moved to a particular successor, then you can use an index (index). The return value (Boolean) is TRUE when the moving was successful and FALSE if the moving failed (successor is occupied, paused, failed). MUs cannot be moved to a source.
<MU-path>.transfer <MU-path>.transfer (<target>) <MU-path>.transfer (<index>)	Transfer moves an MU from one object to another. It moves the entire length to the next object (not just the forefront like the method Move).
<MU>-path.insert(<target>)	Moves and books the MU to the next object.

Within an entrance or exit control, you can access the triggering MU with “@” as the anonymous identifier.

### 3.3.2 Length, Width and Booking Point

All MUs have the properties length, width, and related booking points. The booking point determines the position of the MU; from this, it is booked on the next object and can be accessed from there. The position of the booking point must always be less than or equal to the length or width. Otherwise, you get an error message. In some cases, it is necessary to dynamically adjust the length during the simulation. This may be when the processing is changing the length or if you repeatedly change the direction of transport during the simulation.

**Fig. 3.36** Example Frame

### Example: Change MU Length

We want to simulate a small part of a production. There are steel beams being processed. The beams are initially transported lengthwise, then crosswise, then processed, and then transported again lengthwise. The cross conveyor serves as a buffer. Create a frame according to Fig. 3.36.

Create an entity (“beam”) in the class library. The beam is four meters long and 20 cm broad. Set the booking points for each length and width to 0.1 meters. Redesign the icon of the beam with a width of 80 pixels and a height of five pixels. Delete the old icon and fill the icon with a new color. Set the reference point to the position 40.3. Change the icon “waiting” analogously. Create the following settings: The source creates one beam every three minutes. The lines L1 and L3 are 12 meters long and transport the beams at a speed of 0.05 meters per second. Uncheck the option Rotate MUs in the tab Curve in L2 so that the line L2 transports the beams “cross.” The SingleProc has two minutes of processing time, 75 per cent availability and 30 minutes MTTR. Start the simulation. If a failure occurs in SingleProc, the beams at L2 do not jam, as intended. In the original setting, L2 promotes cross transport, but the capacity is calculated on the length of the conveyor line and that of the beam. If the beam is four meters long, then L2 fits one beam on the line. To correct this, you have to temporarily reduce the length to the width of the beam (from four meters to 20 centimeters). After leaving the cross conveyor, reset the length again to four meters. Therefore, insert two user-defined attributes into the class beam (class library). See Fig. 3.37.

Attributes		Product Statistics		User-defined Attributes					
		New		Edit		Delete			
Name		Value	Type	C.	I.	3D			
mu_length		4	real	*					
width		0.2	real	*					

Fig. 3.37 User-defined Attributes

The method `setLength` must be called twice, once at the entry of MUs in the line L2 (reduce length) and once when MUs enter the SingleProc (set length back to the original value). The method could appear as follows:

```

is
do
  if ?=L2 then
    @.muLength:=@.width;
  elseif ?=SingleProc then
    @.muLength:=@.mu_length;
  end;
end;

```

The beams jam at L2 if the SingleProc is not available.

### 3.3.3 Entity and Container

The entity does not have a basic behavior of its own. It is passed along from object to object. The main attributes are length and width. Booking points determine at which position the entity will be booked while being transported to the succeeding object (mainly track and line). The attribute destination can be used to store the destination station during transport operations (especially during transportation by a worker).

The container can load and transport other MUs. It has no active basic behavior of its own. The storage area is organized as a two-dimensional matrix. Each space can hold one MU. The container is transported from object to object along the connectors or by methods. With containers you can model boxes and palettes. The capacity of the container is calculated by multiplying the x-dimension with the y-dimension. The access to the MUs, which are transported by the container, is analogous to the material flow objects (Table 3.7).

**Table 3.7** Methods for Accessing MUs

Method	Description
<MU-path>.cont	The method "cont" returns an MU that is booked on the container (with the longest length of stay).
<MUpath>.pe(x,y).cont	With "pe" you get access to a place in the container.

### Loading Containers

The approach is somewhat complicated. A container cannot exist by itself. Hence, you need another object that can transport the container—, e.g. to load a box. In addition, transporters can easily transport containers. For loading containers, you can use the assembly station (see chapter on “AssemblyStation”) or the transfer station. You can also load the container using SimTalk methods.

### Example: Loading Containers

We want to develop a method that creates a palette, which is loaded with 50 parts and to pass these parts onto the simulation. Create a frame like in Fig. 3.38.

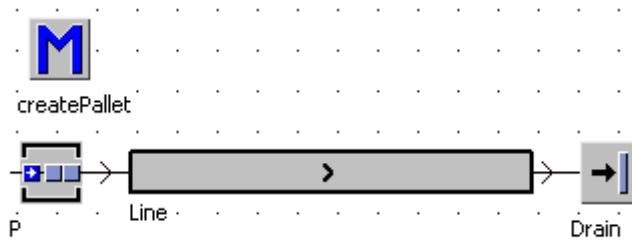


Fig. 3.38 Example Frame

The method `createPallet` must first create the palette on `P` and then create parts on the palette until the palette is full. Create duplicates of the container (Pallet, x-dimension: 25; y-dimension: 2) and entity (part) in the folder MUs.

Method `createPallet`:

```
is
do
  --create palette
  .MUs.pallet.create(p);
  -- create parts
  while not p.cont.full loop
    .MUs.part.create(p.cont);
  end;
  -- pass the palette
  p.cont.move;
end;
```

### Digression: Working with Arguments

You will need the facts described above in the following exercises. Therefore, it would be useful if the method could be applied to all similar cases. For this purpose, you must remove all direct references to this particular simulation from the method (mark bold, italic).

```
is
do
  -- create palette
  .MUs.pallet.create(p);
  -- create parts
  while not p.cont.full loop
    .MUs.part.create(p.cont);
  end;
  -- pass the container
  p.cont.move;
end;
```

The method only contains three specific references:

- Location of the palette (.MUs.pallet)
- Location of the part (.MUs.part)
- Target of creation (p)

These three items are declared as arguments (palette, part, place: object). Next, replace the specific details with the arguments (using find and replace in larger methods).

```
(pallet,part,place:object)
is
do
  -- create palette
  pallet.create(place);
  -- create parts
  while not place.cont.full loop
    part.create(place.cont);
  end;
  -- pass the palette
  place.cont.move;
end;
```

The method Call now just has to include the arguments. Create an init method. The call of the method createPallet in the init method could look like this:

```
is
do
  createPallet(.MUs.pallet,.MUs.part,p);
end;
```

Start the init method. To check whether the method has really produced 50 parts, check the statistics of the palette (double-click the filled palette) on the tab Statistics.

## 3.4 Source and Drain

Using source and drain, you can model the system boundaries of your model. The source of a system represents the feeding of the system with MUs (entities, container and transporter). The drain takes all MUs and destroys them. This prevents the flow of material from accumulating in the system and it can carry out, for example, the throughput measurement of the overall system.

### 3.4.1 Basic Behavior of the Source

The source creates mobile objects (MUs) according to your definition. The source can produce different types of parts in a row or in mixed order. For defining batches and determining the points in time, the program provides different

methods. The source as an active object tries to transfer the produced MU to the connected successor.

### 3.4.2 Settings of the Source

The most important settings are operation-mode, time of creation and MU-selection.

#### Operation-Mode

The operation-mode determines how to proceed with MUs, which cannot be transferred. Blocking means that the generated MUs will be saved (it will produce no new MUs). If you select “Non-blocking,” the source creates another MU exclusively at the time of creation you entered.

#### Time of Creation

The settings in Time of creation determine when you create MUs (Fig. 3.39).

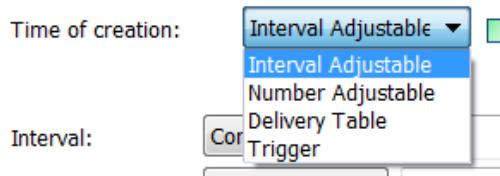


Fig. 3.39 Source: Setting Time of Creation

The default value is **Interval Adjustable**. The production dates are determined by three figures: start, stop, and interval. The first part is produced at the time “Start.” Other parts are produced at an interval. The production of the parts ends with stop. You can enter statistical distributions for all three values.

**Number Adjustable:** Number and interval (a certain number at specified interval) determine the production dates. The settings in Fig. 3.40 will produce 10 parts after 10 minutes of simulation time only once.



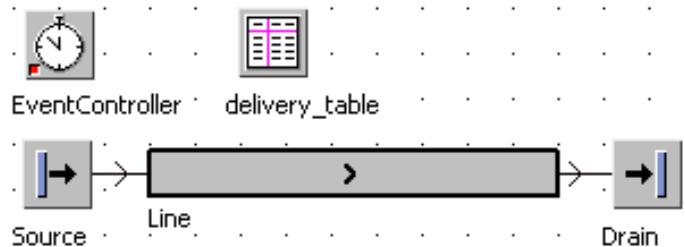
Fig. 3.40 Number Adjustable

**Delivery Table:** The production times and type of parts to be produced are taken from a table (delivery table). Each line in the delivery table contains a production

order. For this purpose, you have to add a table to your frame. If the list is processed until the end, the source stops the generation of MUs.

### Example: Source Delivery Table

Create the frame as per Fig. 3.41. You can change the length of the line by dragging the corner points to the right-hand side.



**Fig. 3.41** Example Frame

Duplicate the object Entity three times in the class library. Rename these duplicates as part1, part2, and part3. The source should produce five parts of type part1 after two minutes, two parts of type part2 after 10 minutes, and four parts of type part3 after 15 minutes. Select Time of Creation—Delivery Table in the dialog of the source. Next, click in the lower part of the window on the button with the three points. Select the table in the following dialog. Finally, click OK. The name of the table will be entered into the dialog of the source. Now open the table in the frame by double-clicks, and enter the contents of Fig. 3.42.

string	Delivery Time	MU	Number
1	2:00.0000	.MUs.part1	5
2	10:00.0000	.MUs.part2	2
3	15:00.0000	.MUs.part3	4

**Fig. 3.42** Delivery Table

The source now produces parts as specified in the table. After the last part has passed the drain, the simulation will be complete.

### MU-Selection

You can see the available settings in Fig. 3.43.

**Constant:** Only one MU type will be produced. Select the path to the respective MU in the dialog (Object Explorer).

**Sequence Cyclical:** Parts are produced according to the sequence you entered in a table (see delivery table). Enter the path to the table in the text box. If the check box “Generate as Batch” is checked, the quantity will be produced at one time. When the sequence is completely produced, the production sequence will repeat.

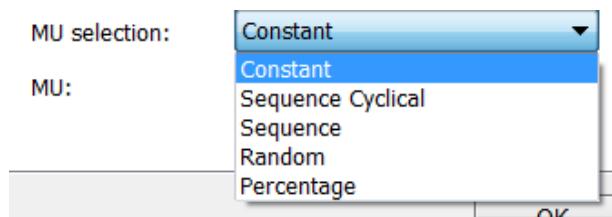


Fig. 3.43 Source: MU-Selection

**Sequence:** See Sequence cyclical; after the end of processing the entries, no repetition will take place.

**Random:** The production is based on a random table.

**Percentage:** The production is based on a distribution table.

#### Example: Randomly Produce MUs

Part1, Part2, and Part3 are to be produced. The ratio is 30 per cent for Part1, 60 per cent for Part2, and 10 per cent for Part3. First, create the parts in the class library. Try to change the color of the parts (right-click—Edit Icons). Create a frame like in Fig. 3.44.

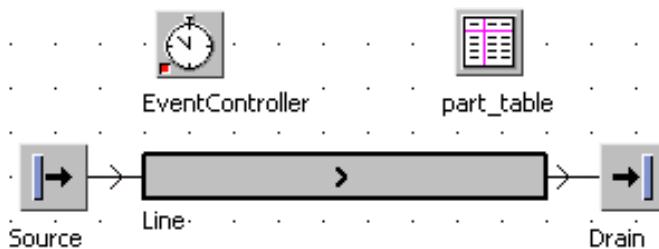


Fig. 3.44 Example Frame

Now select the following settings from Fig. 3.45 in the source.

Interval:	Const	0:03	<input checked="" type="checkbox"/>		
Start:	Const	0	<input type="checkbox"/>		
Stop:	Const	0	<input type="checkbox"/>		
MU selection:	Random	<input checked="" type="checkbox"/>	Stream: 1	<input type="checkbox"/>	
Table:	part_table	<input type="button" value="..."/>	<input checked="" type="checkbox"/>	Generate as batch	<input type="checkbox"/>

Fig. 3.45 MU-selection Random

Confirm your changes with Apply or OK. Open the table by double-clicking it. Enter the data of Fig. 3.46.

	object 1	real 2
string	MU	Frequencies
1	,MUs.part1	0.30
2	,MUs.part2	0.60
3	,MUs.part3	0.10

Fig. 3.46 Parts Table

Start the simulation. The parts will be produced in a “mixed” order.

With sequence cyclical you can easily simulate Just in Sequence deliveries. You should use sequence cyclical only along with the setting blocking (so that the order is not violated).

#### Example: Surface Treatment, Source Sequence Cyclical

A number of different parts run through a chemical process (30 minutes of surface treatment). After that, the parts are processed further on different machines. The simulation flow must branch out after the surface treatment. Create a frame according to Fig. 3.47.

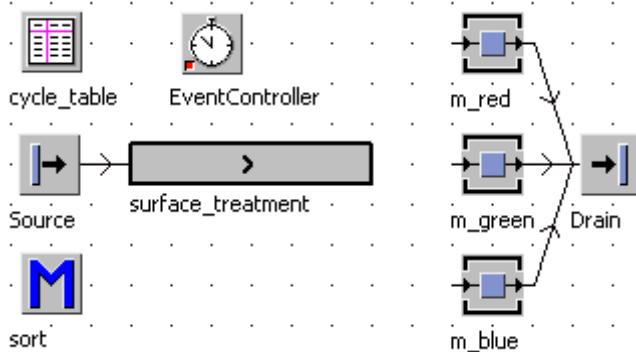


Fig. 3.47 Example Frame

Settings: Source: interval of one minute, sequence cyclical (cycle\_table); a cycle consists of three parts red, two green, and two blue. Create three MUs in the path models: red, green, and blue for the simulation. Color the icons according to the designations. Insert the sequence into the table like in Fig. 3.48 Cycle.

	object 1	integer 2
string	MU	Number
1	.MUs.red	3
2	.MUs.blue	2
3	.MUs.green	2

**Fig. 3.48** Cycle Table

surface\_treatment: length: nine meters; processing time: 30 minutes; M\_red, M\_green, and M\_blue processing time: one minute each; 100 per cent availability. Method sort (exit control (front) of surface\_treatment):

```
is
do
  inspect @.name
  when "red" then
    @.move(m_red);
  when "blue" then
    @.move(m_blue);
  when "green" then
    @.move(m_green);
  end;
end;
```

### 3.4.3 Source Control Using a Trigger

There is the possibility to control sources with triggers. You specify for the trigger a "production table." The trigger then initiates the generation of a certain number of parts at the indicated times. Through the combination of multiple sources, you can model daytime fluctuations or seasonal fluctuations in the number of MUs per time unit.

#### Example: Source Control Using a Trigger

Create a frame according to Fig. 3.49.

Create the following settings: Source: interval of five minutes; check\_in: processing time: 2:20; oldValue: data type integer; initial value: zero. The generator calls getJpH every hour. The source\_peak generates from 11:00 to 12:00 producing an additional five parts per hour, then one hour additional 12 parts per hour and then one hour additional 5 parts per hour. The distances between the individual parts to be distributed randomly (uniform distribution). The getJpH method checks the number of MUs per hour which leave the check\_in. To connect the source\_peak with the trigger, choose in the source the Time of creation—Trigger. Then turn off the inheritance next to the trigger button. Click the Trigger button and enter in the list the trigger name (Fig. 3.50).

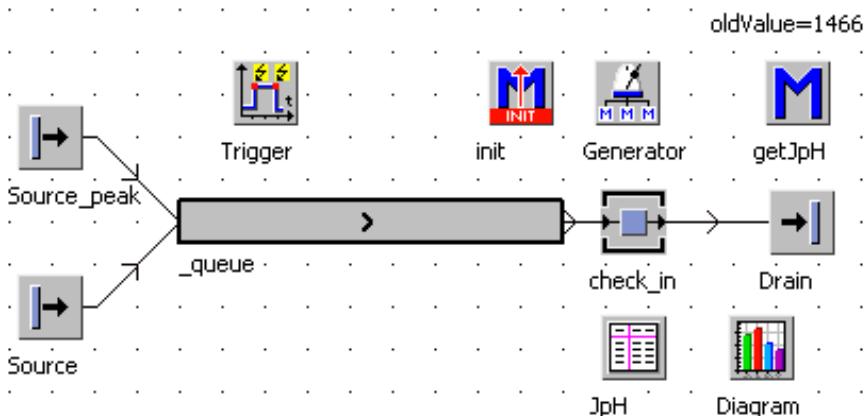


Fig. 3.49 Example Frame

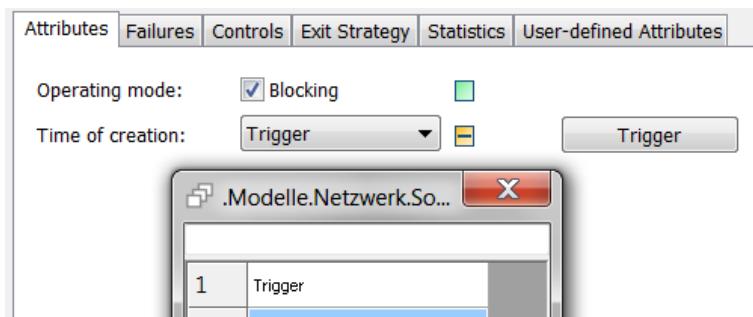


Fig. 3.50 Source—Trigger Connection

In the trigger, you must first define a period length (in the example, one day) and activate repeat periodically (Fig. 3.51).

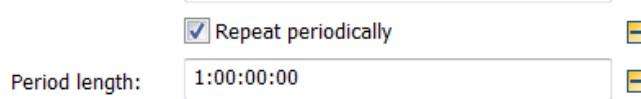


Fig. 3.51 Trigger Setting 1

The information for generating the MUs must be entered in the value table of the trigger (before that, switch off the inheritance of the table). The generation information is defined as a string in the following form:

<count>, <MU-class>, <distribution-type>, <stream> [, <interval-distribution parameter>]. In the example, the definition of the generation might look like Fig. 3.52.

	time 1	string 2
string	Point in Time	Value
1	11:00:00.0000	5,.BEs.part,uniform,1,06:00,18:00
2	12:00:00.0000	12,.BEs.part,uniform,2,04:00,08:00
3	13:00:00.0000	5,.BEs.part,uniform,3,06:00,18:00
.	.	.

**Fig. 3.52** Trigger—Generation of Information

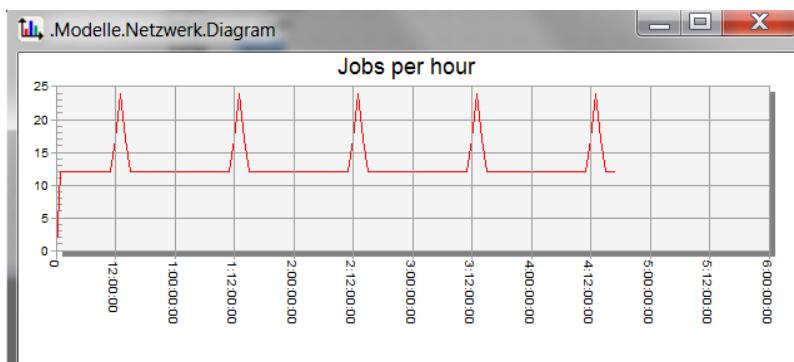
The number of MUs per hour (or jobs per hour, JpH) can be determined using the attribute statNumOut of the check-in. After each hour, determine this using a historical value (old\_value, from the last call) and the current number, and enter it into the table (method getJpH).

```
is
do
    JpH.writeRow(1,Jph.yDim+1,
        eventController.simTime,
        check_in.statNumOut-oldValue);
    oldValue:=check_in.statNumOut;
end;
```

In the init method, you must delete the values from the previous simulation run.

```
is
do
    jph.delete;
end;
```

If you visualize the table on a chart, you can see the different throughputs per hour (Fig. 3.53).



**Fig. 3.53** Diagram JpH

### 3.4.4 User-defined Source with SimTalk

The source has some limitations that could force you to think about a custom solution. You face problems with the source if you do not know exactly when you require parts, need to start and stop the source or the kind of parts to be produced changes during the simulation. It is easy to create your own source. You can try it, for example, with a combination of a method and generator. The following example shows a solution with which you can produce at certain times a certain amount of parts.

#### Example: User-defined Source

The basis of the source will be a buffer. Before you can begin to build the model, duplicate a buffer and name it "specialSource". Create a simple frame (Fig. 3.54). Use the new object specialSource.

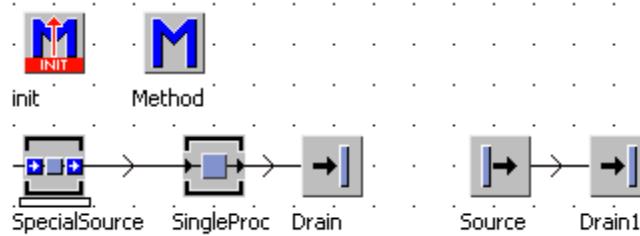


Fig. 3.54 Example Frame

Set the source to a distance of one minute. The example does not generate events in the initial phase. But if the event list of the event manager is empty, Plant Simulation terminates the simulation. To try it anyway, introduce a small "event-bypass." The easiest way is using a source connected to a drain. This is not necessary in larger simulations because enough new events are generated all the time. Create in the class of the SpecialSource user-defined attributes according to Fig. 3.55.

User-defined Attributes						
New		Edit		Delete		
Name	Value	Type	C.	I.	3D	
DeliveryTable		table	*			
init		method	*			
runDelivery		method	*			

Fig. 3.55 User-defined Attributes of SpecialSource

Format the table DeliveryTable like in Fig. 3.56.

	time 1	object 2	integer 3	string 4
string	Delivery Time	MU	Number	Name
1				

Fig. 3.56 DeliveryTable

The method runDelivery should have the following behavior: The method waits until there are entries in the table. Then it reads the first entry, determines the number of parts to be produced and adjusts the capacity of the buffer accordingly. Then, the method generates the correct number of parts. At the end, the method calls itself and waits for the next entry. The method should have the following content:

```

is
  deliveryTime:time;
  MU:object;
  MUobj:object;
  number:integer;
  Muname:string;
  i:integer;
do
  --wait for delivery order
  waituntil self.~.deliveryTable.yDim > 0 and
  self.~.empty prio 1;
  -- read the first line
  deliveryTime:=self.~.deliveryTable[1,1];
  MU:=self.~.deliveryTable[2,1];
  number:=self.~.deliveryTable[3,1];
  Muname:=self.~.deliveryTable[4,1];
  -- delete the entry
  self.~.deliveryTable.cutRow(1);
  -- create enough capacity
  if number > 0 then
    self.~.capacity:=number;
  end;
  --create parts
  waituntil root.eventController.simTime >=
    deliveryTime prio 1;
  for i:=1 to number loop
    if not isVoid(MU) then
      MUobj:=MU.create(self.~);
      MUobj.name:=Muname;
    end;
  next;

```

```
--recursion
  self.methcall(0);
end;
```

The internal init method (custom attribute in the SpecialSource class) has to call the method runDelivery once at the start of the simulation.

```
is
do
  self.~.runDelivery;
end;
```

The init method in the network can now be used to control the behavior of the specialSource. Therefore, include two entries in the DeliveryTable. Method init (on the network):

```
is
do
  SpecialSource.deliveryTable.delete;
  SpecialSource.deliveryTable.writeRow(1,
    SpecialSource.deliveryTable.yDim+1, 0,
    .MUs.Entity, 1, "Task1");
  SpecialSource.deliveryTable.writeRow(1,
    SpecialSource.deliveryTable.yDim+1, 1000,
    .MUs.Entity, 10, "Task2");
end;
```

You can create the entry in the deliveryTable at any time during the simulation.

### 3.4.5 The Drain

The drain is an active material flow object. It has a single place and destroys MUs after processing them. Set the processing time to zero seconds in the drain or to the time a following process would require. The drain collects a number of important statistical data such as throughput, number of destroyed parts, etc. Click on the tab Type Statistics.

## 3.5 Single Processing

Single processing can be modeled easily in Plant Simulation. Plant Simulation holds the MU on the block until the processing time (and possibly recovery time or setup time) is over. Thereafter, the MU is moved to the next block. More interesting is the study of a couple of machines that are linked together using conveying technology.

### 3.5.1 SingleProc, Fixed Chained Machines

The SingleProc accepts exactly one MU from its predecessor. After the setup, recovery, and processing time, the MU will be transferred to one of its successors. While an MU is located on the object, all other newly arriving MUs will be blocked. Only after a successor is free and unoccupied will the MU be transferred (it is possible to define different transfer procedures). You can use it to simulate all machines and jobs that handle one part after another. One part at a time can be located on the workplace or the machine. By combining (conveyor) lines and SingleProcs, it is easy to simulate manufacturing systems that consist of processing stations, which are fixed connected through conveyor systems.

#### Example: Chained Machines

You want to simulate a machine system consisting of two machines. The machines are connected to each other by conveyors, which convey parts from one machine to another. Build a frame as in Fig. 3.57.

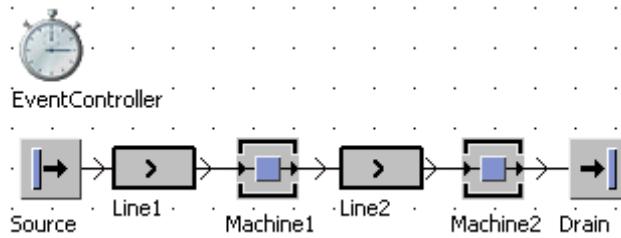


Fig. 3.57 Chained Machines

### 3.5.2 Batch Processing

In such cases for modeling, in addition to the machines for each machine, you need at least two spaces for the containers. The parts are then loaded from the blank part containers on the machine and moved from the machine into the container with the finished parts (if the operator of the machine is not to be simulated). When all parts have been processed, the container with the finished parts will be transported to the next machine and the empty container is moved on the space for the finished parts. You can then begin processing anew after the arrival of the next fully blank container.

#### Example: Batch Production

You are to simulate batch production. Parts are delivered in containers and placed close to the machines. The machine operators remove the parts from the container and place the finished parts onto another container (finished parts). Once the batch is processed, the container with the finished parts will be transported to the next workplace and the empty container will be transferred. Create a frame according to Fig. 3.58.

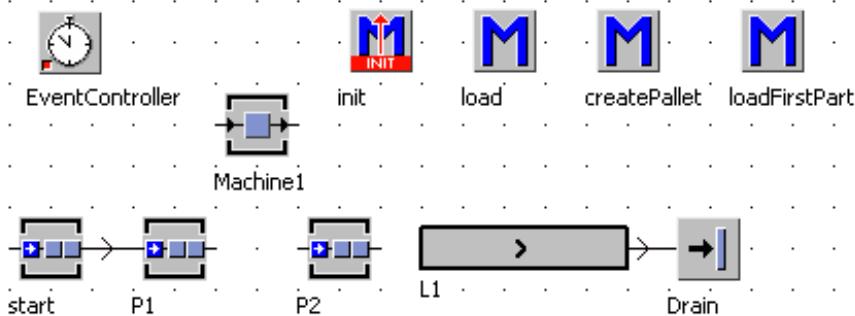


Fig. 3.58 Example Frame

Settings: Start, P1, P2 each at one place with a processing time of zero seconds, L1 of four meters length, speed of 1 m/s, Machine1 processing time of one minute and no failures.

*Digression: Reusing Source Code from Other Models*

We want to use the same method (source code) as in the earlier example “loading of containers.” You can do this in two ways:

1. Save and import the method as an object file.
2. Export and import the methods as text.

Saving and importing the method as an object file:

To do 1), you can store classes and frames as objects at any time. The functionality is located in the class library. Therefore, you must first create a “class” out of the method in the frame. Hold down the Ctrl key and drag the method from the frame to the class library (if you do not hold down the Ctrl key, the method will be moved). If needed, rename the method in the class library. From here, you can save the method as an object. Select the context menu (right mouse button) and select Save Object As (Fig. 3.59 Save ).

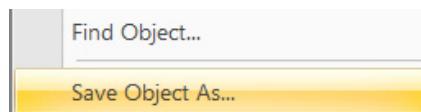


Fig. 3.59 Save Object

You can now load the method into another file as an object file. Right-click on a folder icon. Select Save/Load—Load Object from the context menu (Fig. 3.60).



Fig. 3.60 Load Object

You can now select the file and insert the object into the class library.

**Caution:** When you import the objects, the compatibility of versions/licenses will be considered. You cannot use objects from newer versions of Plant Simulation in older versions (or in eM-Plant). Likewise, you cannot load objects into commercial versions (e.g. Standard or Professional), that were created with a non-commercial license (e.g. Educational, Student, Demo).

To accomplish 2), you can export and import the source code of the method as a text file. Open the method `createPallet`. Select **File—Save** as in the method Editor. Select a location and a name for the file. Insert a method in the new frame (without content; the contents will be completely replaced when you import a text file). Select **File—Open** in the method Editor. Select the text file with the source code, and confirm. The text of the method is completely overwritten by the contents of the file.

### Continuation of Example: Batch Production

If not already available, create the palette container (capacity 50 parts), entity: part. The init method should produce a full palette (50 parts) at the station P1 and an empty palette at the station P2.

```
is
do
  deleteMovable;
  createPallet(.MUs.pallet,.MUs.part,start);
  .MUs.pallet.create(p2);
end;
```

The simulation model (without a dismantle station) could appear as follows. The palette itself transfers the first part of the palette onto the machine (exit control of P1, method `loadFirstPart`).

```
is
do
  --move the first part to the machine
  @.cont.move(Machine1);
end;
```

If a part on the machine is ready, it triggers the exit sensor of the machine (method `Load`). The method transports the part to the palette on the station P2, and a new part from the palette on P1 to the machine. If the finished parts palette is full (p2), they will be transported to F1, the empty palette will be moved from P1 to P2, and a new palette will be generated at the beginning.

Method `load` (exit control front machine):

```
is
do
  -- load the part onto the palette
```

```

@.move(p2.cont);
if p2.cont.full then
  p2.cont.move(L1);
  p1.cont.move(p2);
  -- create a new palette
  createPallet(.MUs.pallet,.MUs.part,start);
else
  -- already parts on p1
  p1.cont.move(machine1);
end;
end;

```

You can access the part on the palette with p1.cont.cont.

In our case, p1.cont is a palette and p1.cont.cont is a part.

#### *Digression: Working with Arguments 2—User-defined Attributes*

Imagine that the simulation of the production contains 50 machines (each with two buffer places). You need to change the method “load” and “loadFirstPart” for each machine. It is much faster to use two methods that fit for all machines. Therefore, some changes are necessary.

First step: Which object in the method has a reference to a specific case?

- P1, P2
- L1
- Machine1
- createPallet (only machine1)

The anonymous identifier “?” can replace Machine1. P1 and P2 are buffers belonging to the machine. If you are using many similar machines, it could be worth your while to define the respective attributes in the class SingleProc (in the class library).

Name the attributes:

- bufferGreenParts (object)
- bufferReadyParts (object)
- successor (object)

Open Machine1 in the frame. In the dialog of the object select: Navigate—Open Origin. This will open the class of the SingleProc in the class library: Click the tab User-defined Attributes, and define the two buffers and the successor (Fig. 3.61).

Name	Value	Type	C.	I.	3
bufferGreenParts	(?)	object	*		
bufferReadyParts	(?)	object	*		
successor	(?)	object	*		

**Fig. 3.61** User-defined Attributes

Save your changes by clicking OK. You can now assign the two buffers (buffer-GreenParts → P1, bufferReadyParts → P2) to Machine1 and F1 as successor. To do this, click the tab User-defined Attributes in the object Machine1, double-click the relevant line and select the buffer or L1 (Fig. 3.62).

Importer		Failure Importer		User-defined	
		New	Edit	Delete	
Name		Value	Type	C.	
bufferGreenParts		P1	object		
bufferReadyParts		P2	object		
successor		L1	object		

**Fig. 3.62** User-defined Attributes Machine1

This makes programming the method easier for many applications:

- machine1 will be replaced by ?
- p1 will be replaced by ?.bufferGreenParts
- p2 will be replaced by ?.bufferReadyParts
- L1 will be replaced by ?.successor

Specific instructions for a machine are written into an “if then else...” statement. Method load:

```

is
do
  -- load the part into the palette
  @.move(? .bufferReadyParts .cont);
  if ? .bufferReadyParts .cont .full then
    ?.bufferReadyParts .cont .move(? .successor);
    ?.bufferGreenParts .cont .move(
      ?.bufferReadyParts);
  -- create a new palette
  if ? = machine1 then
    createPallet( .MUs .pallet, .MUs .part, start);
  
```

```

    end;
else
  -- already parts on p1
  ?.bufferGreenParts.cont.cont.move(?) ;
end;
end;

```

Likewise, you can convert the method `loadFirstPart`. Define an attribute "machine" in the class of the PlaceBuffer (type object). Set `Machine1` as the value for the attribute `machine` in the buffer `P1`. The method should look like this:

```

is
do
  -- move the first part to the machine
  @.cont.move(? . machine);
end;

```

You can now easily extend the simulation without the need to write new methods. You need only assign the methods to the sensors and set the attributes of the objects.

## 3.6 Simultaneous Processing of Several Parts

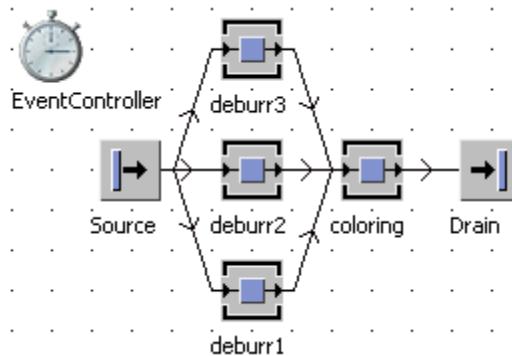
If several parts are machined simultaneously on one machine, you need to analyze the behavior of the machine. The `ParallelProc` in Plant Simulation usually has only limited applicability for these tasks.

### 3.6.1 *The ParallelProc*

The basic behavior of the `ParallelProc` is the same as that of a `SingleProc` with multiple places. Without a control, a newly arriving MU will always be placed on the place that was empty for the longest time. When an MU with a different name arrives, the entire object will be set up. The processing time for each place begins immediately after repositioning MUs on the free place. Each part leaves the parallel station when its processing time is over (individually and not as a bunch).

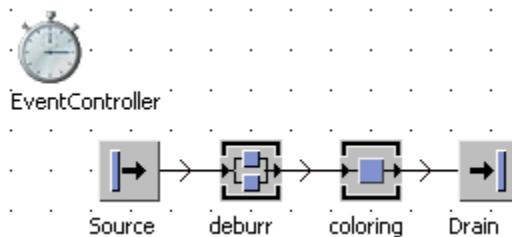
#### Example: `ParallelProc`

After deburring, parts will be treated in a coloring. Since deburring has a long processing time, there are several places for deburring. `Source1` delivers one part every two seconds. The deburring station has three places with a processing time of six seconds per place and part. Coloring takes two seconds. The frame with `SingleProcs` could look like in Fig. 3.63.



**Fig. 3.63** Frame Example: ParallelProc

To simplify the model, use a station with three processing stations (ParallelProc, Fig. 3.64).



**Fig. 3.64** ParallelProc

Set in the tab Attributes the number of places and in the tab Times the processing time per place (six seconds).

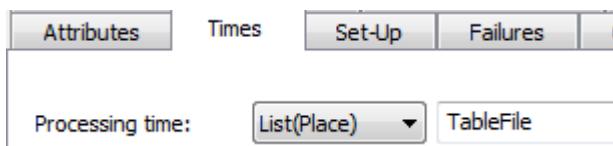
Every "row" takes on x places (x-dimension) in y "columns" (y-dimension). The number of places results from the multiplication of the x-dimension and the y-dimension. If you want to reduce the dimension of the ParallelProc, then it may not find MUs on the places (otherwise, you get an error).

Normally, each place of a ParallelProc has the same processing time. Nevertheless, it is possible to define different times for each place.

Proceed like this:

1. Define the number of places (x-dimension, y-dimension)
2. Insert a TableFile into the frame (folder InformationFlow).

Select within the drop-down list of the tab Time the option Processing time: "List (place)" (Fig. 3.65)



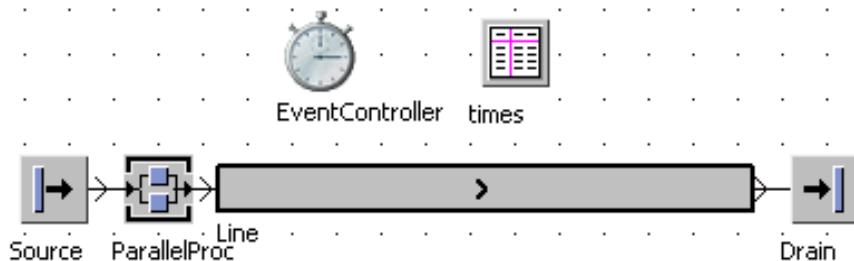
**Fig. 3.65** Different Times for Different Places

3. Enter the name of the table into the field.
4. Enter the processing times for the individual stations into the table (analogous to the position of the places in x- and y-dimensions).

Note: Until Plant Simulation 8.2, you will need to format the table. You have to allocate, according to the x-dimension of the ParallelProc, a number of columns to the data type time. From Version 9 onward, Plant Simulation formats the table according to the dimension of your ParallelProc (x columns and y rows) if the data type of the table columns does not match. As a result, you have to set the x-dimension and y-dimension values for the ParallelProc before assigning the table.

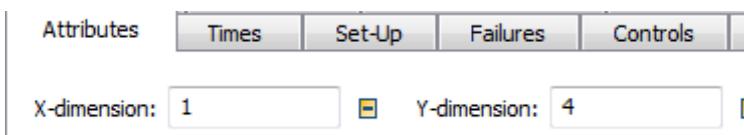
#### Example: ParallelProc; Different Processing Times

A production line has four deburring stations with different technical equipment. For this reason, the individual stations have different processing times: station 1 and station 4 have one minute each, station 2 has two minutes and station 3 has four minutes. Create a frame according to Fig. 3.66.



**Fig. 3.66** Frame Example

Settings: source: interval of 20 seconds; line: length 12 meters, speed 0.08 m/s; drain: zero seconds processing time. To define the processing times, follow these steps: Set the dimension of the ParallelProc (Fig. 3.67).



**Fig. 3.67** Dimension of ParallelProc

Select “List(Place)” from the list Processing Time. Enter the name of the table “times” into the text box (or drag the table from the frame in the field, as in Fig. 3.68).

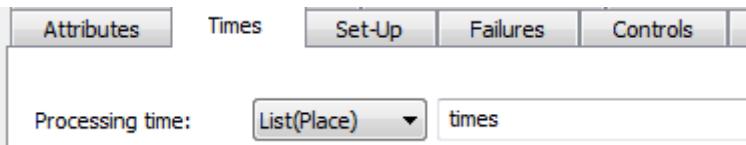


Fig. 3.68 Setting Times

Confirm your changes by clicking OK. Open the table and enter the times (Fig. 3.69).

	time
1	2:00.0000
2	4:00.0000
3	1:00.0000
4	2:00.0000

Fig. 3.69 TableFile times

### 3.6.2 Machine with Parallel Processing Stations

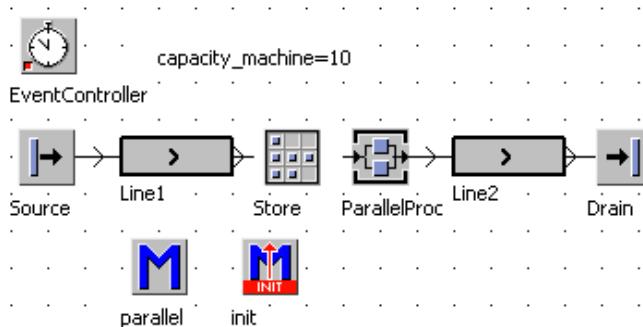
In many cases, the machines that process multiple parts simultaneously show in reality a completely different behavior than the ParallelProc of Plant Simulation. Normally, such machines are loaded with multiple parts. Then begins, for all parts at the same time, the processing; after processing, all parts leave the machine "relatively simultaneously." One approach to simulate this behavior is preventing the transfer of the parts at different times to the machine. For this, you could collect the parts in a buffer (simulating the clamping of the parts on the machine) and relocate them simultaneously to the machine (e.g. by means of a loop). All parts then begin its processing at the same time and the parts can leave the machine simultaneously. This corresponds to the behavior of the real machine.

#### Example: Machine with Parallel Processing Stations

We want to simulate parallel processing. Five parts will be mounted on a machine and then processed together within 20 minutes. After processing, the five parts exit the machine (almost) simultaneously. The machine receives one part every four minutes. The following approach would be possible: The parts will be collected in a store until the required number of parts is reached. If the number of parts is reached, the entrance of the store will be locked. If the machine is empty, then all the parts are moved onto the machine by a loop. After machining of all

parts, the last part (exit control rear) removes the blockage of the entrance of the store. Then, the cycle starts again.

Create the frame from Fig. 3.70.



**Fig. 3.70** Example Frame

Method parallel:

```

is
  i:integer;
do
  if ?=Store then
    -- eventhandling of the store
    if store.numMU = capacity_machine then
      waituntil ParallelProc.operational and
        ParallelProc.empty prio 1;
    -- move all parts
    for i:=1 to capacity_machine loop
      store.cont.move(ParallelProc);
    next;
    -- lock the entrance of the store
    Store.entranceLocked:=true;
  end;
  elseif ?=ParallelProc then
    if ParallelProc.empty then
      Store.entranceLocked:=false;
    end;
  end;
end;

```

Explanation: The control is to be used for two objects. With “?”, you can query which object initiated the call. You can program the control with a respective conditional branch:

```

if ? = Store then
  -- method is called from the store

```

```

elseif ? = ParallelProc then
  -- method is called from the ParallelProc
end;

```

Method init: To ensure that this object can be easily adapted (especially the capacity), capacity\_machine is defined as the global variable. This global variable will be read by the init method, and the capacity of the store and ParallelProc will be adjusted accordingly. You have to also clear the entrance of the store in case the simulation stopped during processing of the ParallelProc.

```

is
do
  deleteMovables;
  Store. entranceLocked:=false;
  -- check and possibly reset the value of
  -- the global variable
  if capacity_machine <1 then
    capacity_maschine := 1;
  end;
  --set the capacity of the ParallelProc
  ParallelProc.ydim:=1;
  ParallelProc.xdim:=capacity_machine;
  --capacity of the Store
  Store.ydim:=1;
  Store.xdim:= capacity_machine;
end;

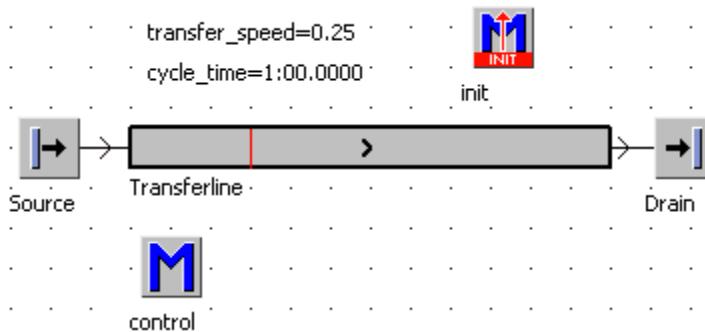
```

### 3.6.3 *Continuous Machining, Fixed Transfer Lines*

There are many cases where parts are processed while they move slowly through the machine (e.g. heat treatment, washing machines, painting). For these cases, you can easily use the (conveyor) Line object. Conventional and fixed transfer lines can also be modeled easily with the Line. If you set the speed of the conveyor line to zero, the conveyor line stops. You can use the MU spacing on the conveyor line to set the distance between the processing stations. You only need a sensor and a small control to achieve the correct behavior.

#### **Example: Fixed Transfer Line**

You need to simulate a fixed transfer line involving three machines. The transfer line has the following behavior: There are always three parts simultaneously in the system. After the required cycle time (one minute), all parts are simultaneously moved to the next machine. Then, the processing on the machines starts at the same time. The machines have to give each other a distance of three meters. The transport is carried out at a speed of 0.25 m/sec. Create a frame according to Fig. 3.71.



**Fig. 3.71** Example Frame

The source produces one part each minute, while the conveyor line has a length of 12 meters and an initial speed of 0.25 m/sec. Transfer\_speed and cycle\_time are global variables (data type: real and time). Add a sensor at the position of three meters in the transferline, and assign the control as sensorcontrol. The control is relatively simple. If the MU triggers the sensor, the transferline is stopped. After the elapse of the cycle time, the transferline is reset to the initial speed. Method control:

```

(SensorID : integer; rear : boolean)
is
do
    transferline.speed:=0;
    wait(cycle_time);
    transferline.speed:=transfer_speed;
end;

```

If you stop the simulation when the conveyor line is stopped (speed = 0), then this setting will be retained when you restart the simulation. Therefore, it is necessary to set the line speed on the initial value during initialization of the simulation. Method init:

```

is
do
    transferline.speed:=transfer_speed;
end;

```

The behavior can be observed best when you select the real-time option in the EventController.

### 3.6.4 The Cycle, Flexible Cycle Lines

With the cycle object, you can synchronize a set of objects. The MUs will be passed on only if all stations in the balanced line have finished processing, neither failed nor paused, and the next station is ready to receive MUs. Connectors must

link the stations of the balanced line. Enter the first and last objects of the balanced line into the cycle object. The following example demonstrates how the cycle object works.

### Example: Cycle

Three machines will be synchronized. Create the frame from Fig. 3.72.

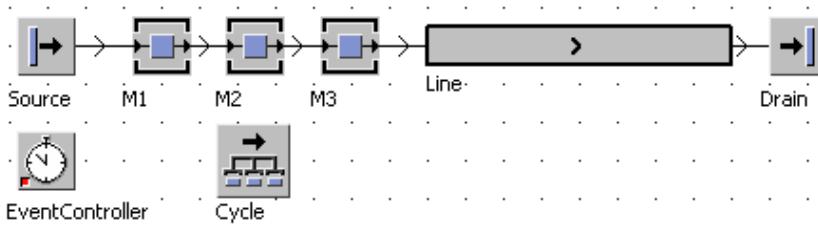


Fig. 3.72 Example Frame

Settings: source: interval of two minutes; M1: processing time of two minutes; M2: processing time of two minutes; M3: processing time of one minute; Line: length of eight meters, time of eight minutes. Let the simulation run. The station M3 is finished earlier than the other stations and remains empty for a while. Now double-click the cycle object. Enable the cycle object with Active and enter the first and the last stations (Fig. 3.73).

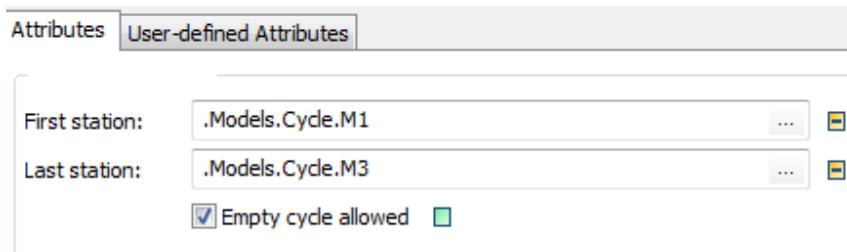


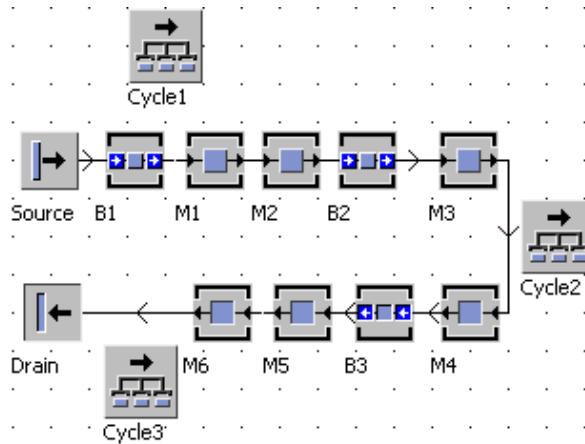
Fig. 3.73 Cycle: Tab Attributes

The part on the object M3 remains on the station until all other stations are finished too. With the help of the cycle module, you can simulate both, fixed and flexible cycle lines. In a flexible cycle line, buffers are used to decouple the sections of the line and, thus, to increase the overall availability of the system. Minor problems of the sections are compensated for by the buffers. The following example demonstrates this.

### Example: Flexible Cycle Line

A cycle line consists of six machines. The line is separated in sections. Each section consists of two machines. The sections are decoupled with buffers. Each buffer has a capacity of 100 parts and a process time of three seconds. The cycle

time is 45 seconds. Each machine has an availability of 95 per cent and 15 minutes MTTR (set for each machine a different random stream). Create a frame according to Fig. 3.74.



**Fig. 3.74** Example Frame

Assign to any two machines one cycle object and activate them. Let the simulation run for 10 days and note the output (e.g. drain, tab Statistics—entries). Set in a second run, the capacity of all buffers to one place and the processing time of the buffers to zero seconds (so that you disable the buffer and a failure of one individual machine has a direct impact on the total output). The throughput will be considerably worse.

### 3.7 Assembly Processes

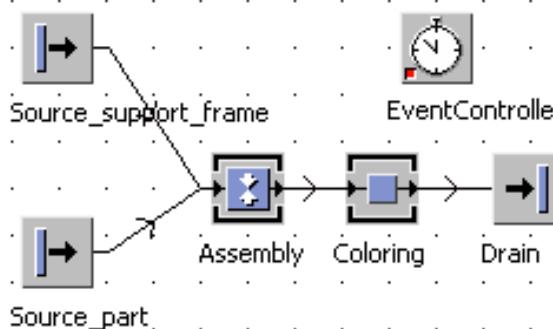
For the modeling of assembly processes, there is a special object in Plant Simulation (AssemblyStation). If necessary, assemblies can be easily modeled with SimTalk. Define the assembly concept more comprehensively. All processes in which an MU is loaded in or on a different MU can be represented by means of the AssemblyStation. This includes the loading of transporters and the packaging of parts.

#### 3.7.1 *The AssemblyStation*

The AssemblyStation adds mounting parts to a main part or simulates assembly processes by destroying the single parts and generating the assembled part. The AssemblyStation facilitates the simulation of assembly operations.

### Example: Assembly

Before coloring a part, it will be mounted onto a support frame. In the absence of the support frame, the coloring is not possible. The assembly lasts two minutes; coloring also takes two minutes. Main part: support\_Frame (container), mounting part: part (entity). Create the frame from Fig. 3.75.



**Fig. 3.75** Example Frame

Make sure that you first connect the source\_support\_frame, then source\_part with the assembly. You have to convince the support\_source\_frame to generate support\_Frames (default is Entity). Double-click on the source and select MU Container. Create the following settings in the AssemblyStation. **Assembly table:** Select the parts that you want to assemble, according to different points of view:

- MU types
- Predecessor number

If you do not select an assembly list, one of each part will be assembled.

Select Predecessors and open the predecessor table (button Open): Enter the number of the predecessor and the amount of assembled parts into the list. If you select the assembly mode “Attach MUs,” do not enter the main part into the list. In the example above, one part of predecessor 2 is to be mounted: Select Assembly table with—Predecessor, and then click Open. Fill the list like in Fig. 3.76.

	Predecessor	Number
1	2	1

**Fig. 3.76** Assembly Table

If you select MU-types, enter the name of the MU-class and the respective number of parts into a table.

**Note:** You can show the numbers of the predecessors. Select View—Options—Show Predecessors in the frame window.

**Main MU from Predecessor:** Define which workstation provides the main part. The number is derived from the sequence in which you established the connections. Please note that the main MU itself must be able to accept parts (e.g. container) if you select the option Attach MUs.

**Assembly mode:** You can "load" parts on a main part (the main part must have sufficient capacity—e.g. as container) or destroy all parts and create a new part (assembly part).

**Exiting MU:** The main part (with the loaded components) or a new part can be moved from the assembly. If you create a new part, you will have to select it.

### 3.7.2 Assembly with Variable Assembly Tables

Under certain circumstances, the assembly tables of main parts can be different. Such tasks occur, for example, in order picking or during assembly activities with variants. In such cases, you can reallocate the assembly lists (Bill Of Material, BOM) before starting the assembly activity for each assembly station. If you have connected the sources of the parts to the assembly station, then the assembly station "pulls" the right parts from the predecessors.

#### Example: Assembly with Different BOM

The following example parts are to be assembled according to a BOM. Insert into the class library three entities (red, blue, yellow). Color the entities according to their names. Set the container to a capacity of six. Create a frame like in Fig. 3.77.

Set the sources so that the source\_red produces the part red, etc., the source\_container produces containers at intervals of one minute. Create the following settings in the assembly station: assembly table: MU types; assembly mode: Attach MUs; exiting MU: Main MU; processing time: 1:00 (Fig. 3.78).

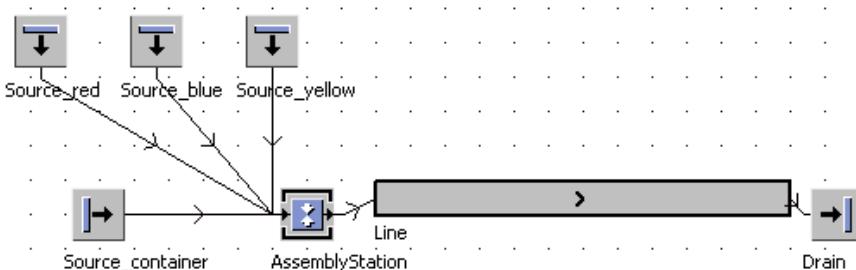


Fig. 3.77 Example Frame



**Fig. 3.78** AssemblyStation Settings

Define for the Container in the class library a custom attribute BOM (table) and then format the table according to Fig. 3.79.

	string 1	integer 2
1	red	
2	yellow	
3	blue	

**Fig. 3.79** User-defined Attribute BOM

To demonstrate the function, the number of red, blue and yellow parts should be distributed randomly, so that the containers show different loads. Assign an entrance control to the source and enter the following programming:

```

is
do
  --dice bom values
  @.bom[2,1]:=round(z_uniform(1,0,6));
  @.bom[2,2]:=round(z_uniform(2,0,(6-@.bom[2,1])));
  @.bom[2,3]:=6-@.bom[2,1]-@.bom[2,2];
end;

```

You can change the assembly list of the AssemblyStation with the method `<path>.assemblyList`. To do this, insert an entrance control in the assembly station, enable the option "Before Actions." Enter the following code into the entrance control:

```

is
do
  --use bom as assembly-list
  ?.assemblyList:=@.bom;
end;

```

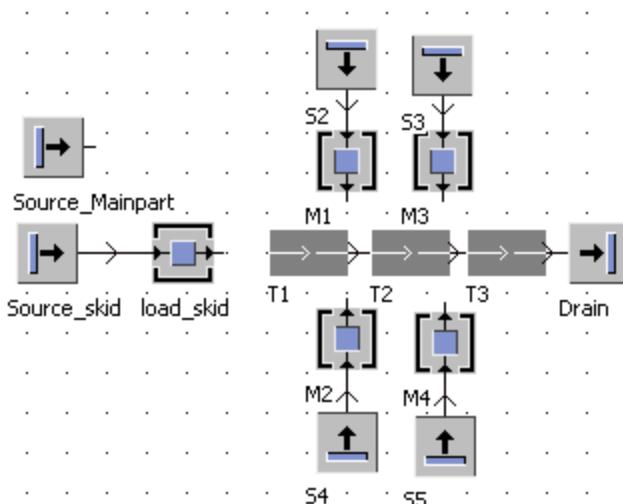
Each container is now loaded with the number of parts specified in the BOM.

### 3.7.3 Use SimTalk to Model Assembly Processes

The use of the AssemblyStation involves some constraints, which necessitate an alternative solution in certain cases. For example, you have to move the main part using connectors to the AssemblyStation as well as the parts to be assembled. Often, the assembled parts are themselves transported on skids or workpiece carriers. When you simulate such assembly lines, you need a huge amount of assembly stations and transport sections to simulate the individual assembly steps. In these cases, you can simulate assembly operations more easily by moving the parts to be assembled, or you can destroy the parts by means of SimTalk and create a new assembly.

#### Example: Assembly with SimTalk

You should simulate a small area of an assembly line. Parts are mounted on a base part. The base part is transported on a skid from station to station. The assembly of the parts is carried out from two sides simultaneously. The cycle of the line is 60 seconds. Create a frame according to Fig. 3.80.



**Fig. 3.80** Example Frame

The source\_Mainpart delivers every minute a container with a capacity of at least four parts. The source\_skid delivers a transporter every minute. Create an exit control for the workplace load\_skid (user-defined attribute exitControl, type method, link the exitcontrol to this method). The sources S2 to S5 each generate parts (Entities). Each "assembly block" (M1–M4) has an exit control (also as a user-defined method attribute). Create for M1, M2, M3 and M4 an init method (for initialization), a custom attribute assemblyFinished (Boolean), assemblyTime (time, 30 seconds) and an attribute to link to the conveyor system (conveyor type Object) like in Fig. 3.81.

Name	Value	Type	C.	I
assemblyFinished	false	boolean	*	
assemblyTime	30.0000	time	*	
conveyor	(?)	object	*	
exitControl		method	*	
init		method	*	

Fig. 3.81 User-defined Attributes

Create best an object M in the class library (duplicate of SingleProc) and modify the settings. Create also a duplicate of the track and call it T. The track needs the following custom attributes: exitControl (method), init (method), inPosition (Boolean) assembly1 (object) and assembly2 (object). Assign the exit control in the tab controls. All blocks that are repeatedly used should be developed in this example so that no adjustment of the source code of the methods is necessary for extending the model. In the exit control of the object load\_skid, the skid waits until a base part is present (source\_mainpart.occupied=true), then the base part will be loaded onto the skid. If the following track is empty or the skid is in motion (running, inPosition = false), then the skid is moved. The following SimTalk code is necessary:

```

is
do
  waituntil source_mainpart.occupied prio 1;
  source_mainpart.cont.move(@);
  waituntil t1.leer or t1.inPosition = false prio 1;
  @.move(t1);
end;

```

Basic considerations for the control assembly and skid: To avoid having to program each assembly station and each skid section separately, all the variable parts are stored as attributes of the objects. These are in the assembly station M1–M4, a reference to the corresponding skid track (conveyor) and a variable that indicates that the assembly process is finished (assemblyFinished). With the help

of an internal init method you can set the variable assemblyFinished to false, and unlock the entrance of the assembly station, Method M.init:

```
is
do
  self.~.entranceLocked:=false;
  self.~.assemblyFinished:=false;
end;
```

Self.~ is a reference to the superior object of the method (~ corresponds to the method location). In the track segments, we need references to the corresponding assembly stations (assembly1 and assembly2) and a status variable, which indicates that the skid has stopped and the assembly can start (inPosition). The development of the methods can now be made without direct object reference in the class library (the content of methods is inherited by default in the derived objects). The assembly stations must wait until the skid is in position. Then, a part can be moved (assembled) to the base part on the skid. To avoid transferring another part, it is better to lock the entrance of the assembly station for the following parts (especially if the assembly stations have different cycle times). This could look like the following (exit control Mx):

```
is
do
  self.~.entranceLocked:=true;
  waituntil self.~.conveyor.inPosition prio 1;
  @.move(self.~.conveyor.cont.cont);
  wait(self.~.assemblyTime);
  self.~.assemblyFinished:=true;
  --wait for the next cycle
  waituntil self.~.conveyor.inPosition = false prio 1;
  self.~.entranceLocked:=false;
  self.~.assemblyFinished:=false;
end;
```

The skid, in turn, must wait until all assembly operations are complete (have been reported as finished). Then the skid is moved to the next segment when it is empty or the next skid is already in motion. Exit control of the tracks:

```
is
  suc:object;
do
  self.~.inPosition:=true;
  suc:=self.~.succ(1);
  waituntil self.~.assembly1.assemblyFinished and
  self.~.assembly2.assemblyFinished and
  (suc.empty or suc.inPosition = false) prio 1;
  @.move;
  self.~.inPosition:=false;
end;
```

The init method of the track segments sets the attribute inPosition to false.

```
is
do
  self.~.inPosition:=false;
end;
```

Now, you only need to set the object references in the blocks and can extend the example to any size without changing the SimTalk programming.

## 3.8 Dismantling

Analogous to the assembly, Plant Simulation provides a standard element for the modeling of dismantle processes. Dismantling processes may also denote the unloading of parts from a vehicle, unpacking of containers or cutting processes.

### 3.8.1 The DismantleStation

The DismantleStation dismantles added parts from the main part or creates new parts. It facilitates the modeling of dismantling operations or unloading processes. In addition, processes can be modeled, in which a part is broken down into several parts, thereby themselves preserved (for example, when punching) or themselves completely broken down (such as when sawing or cutting).

#### Example: DismantleStation

A machine is loaded with a palette (12 parts). The machine unloads the parts at a fixed position with an internal loader from the palette and loads the parts into the machine. After completion, the parts are stored on a different palette at a second position. The palettes are transported to the machine on a three-meter-long conveyor. Behind the machine is available another three-meter conveyor for

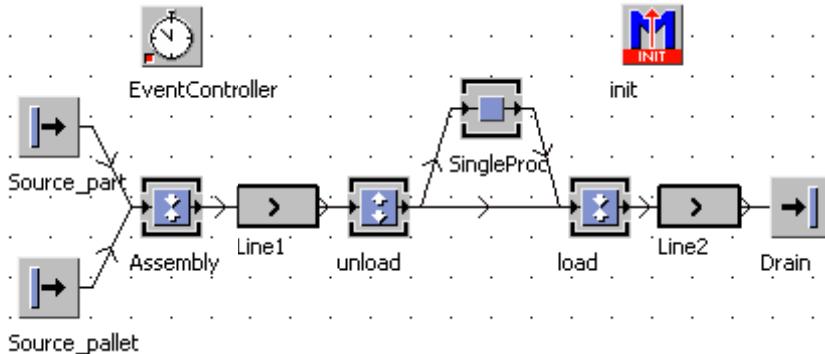


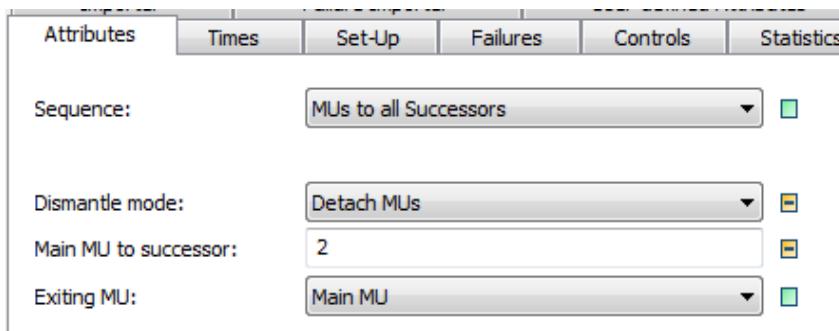
Fig. 3.82 Example Frame

palettes with finished parts. Unloading of the parts can be easily realized with a DismantleStation (the parts are unloaded onto the machine; the empty palettes are transferred to the place where the finished parts will be loaded). Loading the parts on the palette can be realized with the AssemblyStation. Create a part (entity) and a palette (container) in the class library. Enter a capacity of 12 (x-dimension: 3, y-dimension: 4, length and width each 0.5 meters) into the dialog of the container. Create a frame according to Fig. 3.82.

Make the following settings: Source\_part: it will generate entities (part), interval: 1:05; Source\_pallet: it will generate containers (pallet), interval: 12:00; AssemblyStation: processing time: zero, main part: pallet, assembly mode: attach MUs, 12 parts from predecessor 2 (connect first Source\_pallet, then Source\_part with the assembly station); line1: three meters, speed 1m/s; line2: three meters, speed 1 m/s; machine: one minute processing time, loading: main part from unloading (connect at first), seconds processing time, assembly mode attach MUs, 12 parts from machine. To start the simulation, an empty palette must be ready at the loading station. The easiest way is to create an empty palette on line1. The DismantleStation reaches for the empty palette next to the loading station. To ensure that the assembly station receives the palette on the right connector, you need a method object (folder InformationFlow). Rename the method object to init (this method is called when you click init in the EventController). Content of the init method:

```
is
do
  .MUs.pallet.create(line1);
end;
```

Prepare the settings according to Fig. 3.83 in the DismantleStation.



**Fig. 3.83** Setting DismantleStation

**Sequence:** Select how the DismantleStation distributes MUs to its successors from the drop-down list sequence. MUs to all Successors: If you have chosen the option Create MUs in the drop-down list Dismantle mode, the DismantleStation

creates a new MU for each successor and transfers it to the successor. If you have chosen MUs detach from the drop-down list Dismantle mode, the DismantleStation then distributes the MUs to its successors. Note that the DismantleStation transfers the main part to the successor with the number you have entered into the field "Main successor to MU". For the following three menu commands, Plant Simulation requires entries into the dismantle list. MUs exiting independent of other MUs: Each MU is trying to move as soon as possible to the given successor. Main MU after other MUs: The mounted parts exit the DismantleStation first, followed by the main part.

This setting is important if you simulate unloading parts. The empty palette exits empty at the end of the DismantleStation and not before—e.g. if the individual parts cannot be delivered to the successor fast enough.

### Example: DismantleStation; Exit Sequence

Palettes have to be loaded and unloaded. The parts are weighed after unloading and then destroyed by a drain. Weighing takes five seconds. Create a frame like in Fig. 3.84.

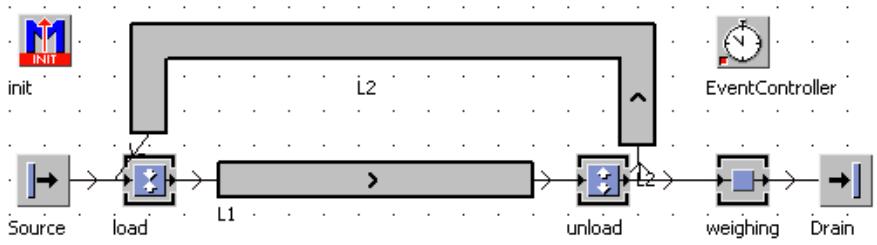


Fig. 3.84 Example Frame

Settings: The source creates one part every four seconds (Entity). First Connect L2, and then the source with the station load. Set the capacity of the container in the class library to 10. Set the assembly station so that 10 parts will be loaded onto the main part (from L2). Assembly and dismantling take no time (processing time 0 seconds). L1 and L2 each have a speed of 1 m/s. First connect the station unload with L2 and then unload with weighing. The init method should create two containers on line L2. Complete the init method as follows.

```
is
do
    .MUS.Container.create(L2, 2);
    .MUS.Container.create(L2, 4)
end;
```

Select the settings from Fig. 3.85 in the DismantleStation (unload).

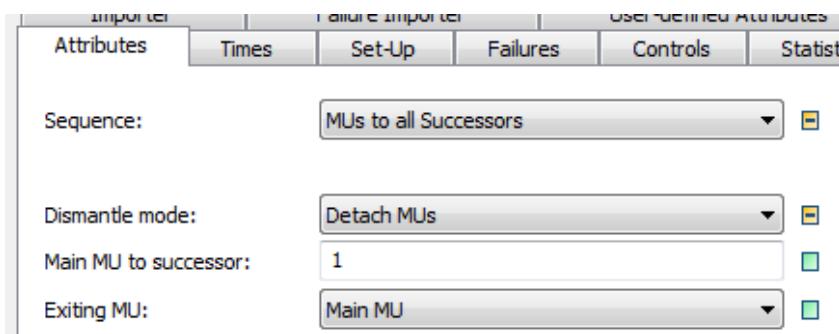


Fig. 3.85 Setting DismantleStation

The palette exits the dismantle station before the last part. In most cases, such behavior does not fit reality. The container must exit the DismantlingStation after the single parts. To do this, change the settings of the DismantlingStation according to Fig. 3.86.

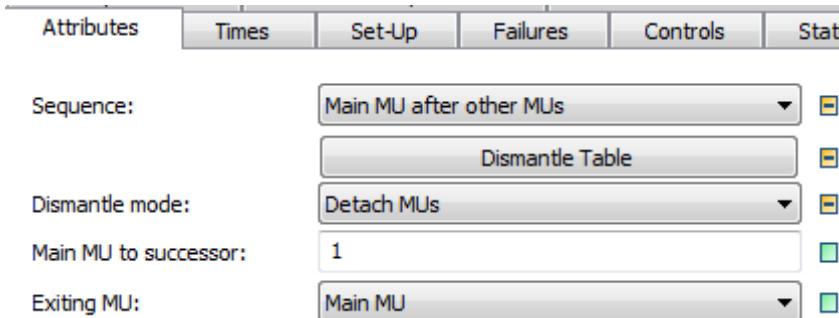


Fig. 3.86 Setting DismantleStation

In the dismantle table, define how many of which parts are to be transferred to any successor (Fig. 3.87).

	MU	Number	Successor
1	.MUs.Entity	10	2

Fig. 3.87 Dismantle Table

Now, all parts will be unloaded first, before the main part is transferred.

**Dismantle Mode:** Select how Plant Simulation deals with mounted parts. It provides two modes: **Detach MUs:** Plant Simulation unloads the mounted parts

from the main part and transfers them to the successors, which are contained in the dismantle table. Create MUs: The DismantleStation creates new parts.

**Main MU to Successor:** This successor number may not be contained in the dismantle table (error message).

**Exiting MU:** Set how Plant Simulation handles the main parts.

### 3.8.2 *Simulation of Split-Up Processes*

You can use the DismantleStation to model split-up processes (e.g. cutting). Destroy the main part and generate the required number of new parts.

#### Example: Split-Up Processes

In the following example, a part (container) is separated into five smaller parts. Create a simple network as shown in Fig. 3.88.

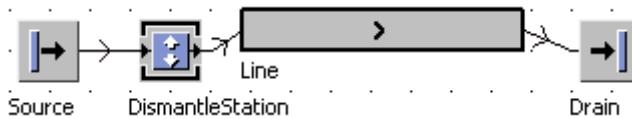


Fig. 3.88 Example Frame

The source generates a container every 15 seconds.

To produce five identical (new) parts, you need to create 4 (5 - 1) parts with the dismantle table and the fifth part as a new main part. In this way, you also prevent the original part (in our case, the container) from leaving the DismantleStation. Prepare the settings as shown in Fig. 3.89 Setting DismantleStation in the DismantleStation.

Attributes	Times	Set-Up	Failures	Controls	Stats
Sequence:	Main MU after other MUs				
Dismantle mode:	Dismantle Table				
Main MU to successor:	Create MUs				
Exiting MU:	1				
MU:	New MU				
	*.MUs.Entity	...			

Fig. 3.89 Setting DismantleStation

Enter according to Fig. 3.90 Dismantle Table, the number of MUs to be generated in the dismantle table (the number reduced by one).

	MU	Number	Successor
1	.MUs.Entity	4	1

Fig. 3.90 Dismantle Table

For the last part to be generated, select in Exiting MU, the option New MU and enter in the field MU the path to the MU to be generated.

### 3.8.3 Dismantle Processes Using SimTalk

Unloading can be modeled using the DismantleStation. For some tasks, however, the simulation may be too complicated and the number of blocks you need for modeling increases enormously. The unloading of pallets can be programmed with relative ease using SimTalk. Access to the pallets and the parts in the palettes occurs through the "underlying" element.

#### Example: Saw, Dismantling with SimTalk

You are to simulate the following process. A block with an edge length of 40 cm is to be sawed into 16 parts. Ten parts each will then be packed into a box. Ten boxes are packed in a carton. Between the saw and the individual packing stations, lines with a length of five meters will be set up. Create the folder "Saw" below models. Duplicate all required classes in this folder. Create the frame shown in Fig. 3.91.

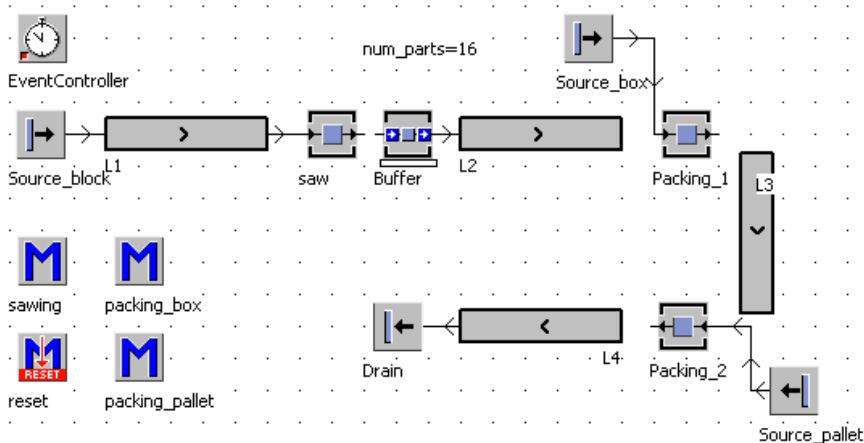


Fig. 3.91 Example Frame

Settings: L1, L2, L3 and L4: length five meters, speed 1 m/s; saw: processing time 10 seconds; packing\_1 and packing\_2: two seconds processing time; drain: zero seconds processing time. Create two entities (block, part) and two containers (box, palette). Block: 0.4 meter length, part 0.1 meters length. Change the icon of the part to a size of 7x7 pixels. Box: container, capacity 10 parts; palette: container, capacity 16 boxes. Arrange your sources so that they produce the correct type of MU (e.g. source\_box). Interval palette: 1:40; interval block: 0:10.

Method Sawing (exit control front of SingleProc saw): The method must destroy the block and create a certain number of parts. Creating the parts works best with a buffer object. The processing time of the sawing can be considered as the processing time of the SingleProc. To make the simulation more flexible, define the number of parts outside the method (e.g. in a global variable in the example: num\_parts). Method Sawing (exit control front saw):

```
is
  i:integer;
do
  -- destroy block
  @.delete;
  -- create num_parts
  for i:=1 to num_parts loop
    .Models.saw.part.create(buffer);
  next
end;
```

At the end of line L2, the parts are to be packed into boxes. If a box is placed on the station Packing\_1, the incoming part is transferred to the box. If the box is full, it will be transferred to line L3. Method packing\_box (exit control L2):

```
is
  box:object;
do
  -- wait for a box
  waituntil packing_1.occupied prio 1;
  box:=packing_1.cont;
  -- pack parts into the box
  @.move(box);
  if box.full then
    box.move(L3);
  end;
end;
```

Similarly, you need to program the method packing\_palette. To ensure a smooth start of the simulation, destroy the MUs when resetting the simulation.

## 3.9 Scrap and Rework

Scrap and rework represent branches in the material flow. Depending on the value of a property or with the help of a random distribution, the successor of the individual MU in the material flow must be determined dynamically. There are at least three different ways to achieve this. You can use the element FlowControl, employ the integrated exit strategy of material flow elements or write your own method (e.g. as exit control).

### 3.9.1 The FlowControl

The FlowControl itself does not process MUs. The FlowControl is always positioned between two or more other objects and defines the flow behavior between these objects. If need be, you can also combine several FlowControl objects.

#### Example: FlowControl, Scrap 1

You are to take a scrap rate at a workplace into account. Scrap represents a branch in the flow of materials (e.g. the quality control sorts a defective part, the defective part is moved to the drain). To simplify the presentation of scrap, the parts have the property "io" with a value of true or false in the simulation. You can branch the flow according to these values. Create a frame like the one shown in Fig. 3.92.

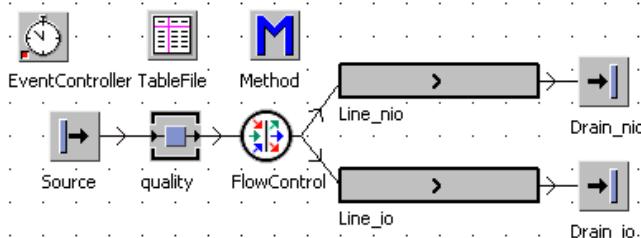


Fig. 3.92 Example Frame

**Quality (quality assurance):** one minute processing time; lines: three meters, speed 1 m/s. The order in which you insert the connectors determines the number of successors. First, connect the FlowControl with the line<sub>io</sub>, then the FlowControl with line<sub>nio</sub>. Duplicate an entity and name it "part". Create a user-defined attribute for the part: name: io, data type: Boolean. Ten per cent of the parts should have the value "false" for the attribute "io." The remaining parts will receive the value "true." The allocation will be done randomly. For the assignment, you can use the source. In the dialog of the source select the option MU-Selection—Random. Select the TableFile for the allocation. Open the TableFile by double-clicking it. Enter the values from Fig. 3.93 into the table.

	object 1	real 2	string 3	table 4
string	MU	Frequencies	Name	Attributes
1	.MUs.part	0.90		attribute
2	.MUs.part	0.10		attribute

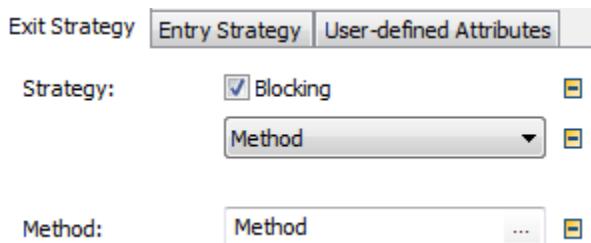
**Fig. 3.93** TableFile

Enter the same part twice. Enter a name in the column Attributes (in the above case, "attribute" is used as an internal name); Plant Simulation then creates a sub-table. Press the F2 key in the field Attributes. It opens another window. Enter the name of the attribute (io) and the value (true/false) into the field with the correct data type (Boolean) see Fig. 3.94.

	string 1	integer 2	boolean 3
string	Name of Attribute		
1	io		true

**Fig. 3.94** Sub-table Attributes

Proceed as described with the second row for the part (Boolean false). You can now evaluate the attribute io in the FlowControl. Select Method on the tab Exit Strategy and select Method (Fig. 3.95).

**Fig. 3.95** Exit-Strategy FlowControl

Within the specified method, you can access the respective part with @; for the successor to be moved, an integer has to be returned (1 or 2 in this example).

Enter the following source code into the method:

```
(r : integer) : integer
is
do
  if @.io=true then
    return 1;
```

```

else
    return 2;
end;
end;

```

Run the simulation and check the results using the type-specific statistics of the drains.

In the tab Exit Strategy you set the exit behavior of the FlowControl. Blocking means that if the successor cannot receive parts, the FlowControl waits until it can receive parts again. Here is a short selection of the strategies:

**Start at Successor 1:** The flow control attempts to always pass the MU on to successor 1. If successor 1 is always receptive, each MU will move to it. The MU will be passed on to the next successor only if moving is not possible (faulty, occupied).

**Cyclic:** The FlowControl tries to move the MU based on the recent passing on the next object (in the list of successors).

**Selection:** The FlowControl tries to move the MUs on to the successor that meets a certain property.

You can also use a method for distributing the MUs: Specify a method that returns the number of the successor. You can access the MU, which is to be transferred with @. If the part cannot be moved to the designated successor, the method will be called again. You can select a percentage distribution. The basis for this is a distribution table. In this table, you enter the percentage for each successor.

**Random:** You can define a distribution function for the transfer. The distribution function is used to determine the successor.

If you select **Cyclic sequence**, then the MUs will be passed in a defined sequence to the successors. The order is entered in the corresponding table.

The distribution **To all Successors** creates duplicates of MUs. Any successor will receive a duplicate (always blocking).

**Assignment:** There is only one successor. You can set a property of the MU using a method.

**MU Attribute:** Here is the successor chosen by the value of an attribute of the MUs. Under the tab Entrance Strategy, set the reunification strategy of the FlowControl (several predecessors).

### 3.9.2 Model Scrap Using the Exit Strategy

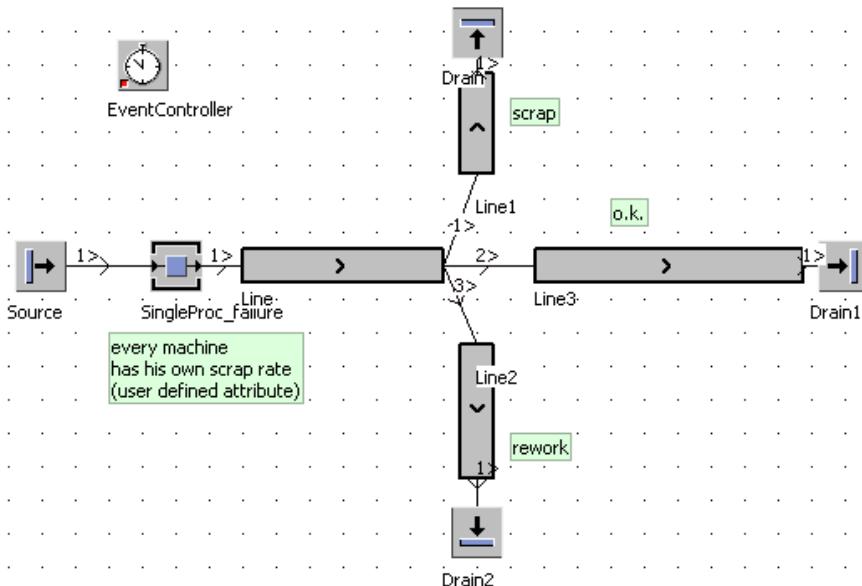
The material flow components themselves have many options from among multiple successors with the help of their exit strategy function. The basic procedure is analogous to FlowControl. You need to mark the MU for scrap or rework in the simulation (e.g. you can use a "Flag"). According to the marking of

the MUs, the material flow must be branched at corresponding positions. Following this, we develop a scrap and rework rate as a property of a SingleProc.

### Example: Machine-specific Scrap Rate

In mechanical manufacturing, you should simulate the effect of increased scrap and rework rate. After each production step, the parts are checked and fed to different processes. In this context, develop the scrap rate and rework rate as properties of the SingleProc. Model the scrap and rework rates as follows: Each part has a characteristic attribute quality (integer). Depending on the machine-specific rate, the value of quality of the part is set to 0 (io), 1 (rework) or 2 (scrap). Then, use the exit strategy of the stations to branch the material flow. Duplicate a SingleProc and Container in the class library. Create the following network. Add the duplicate of the SingleProc in a frame similar to Fig. 3.96.

Insert into the duplicated SingleProc in the class library three user-defined attributes: scrap\_rate (real, value 0.1), rework\_rate (real, value 0.4) and processing\_result (method). Set the method processing\_result as the entrance control of the SingleProc. Insert in the Entity a user-defined attribute quality (integer). Set the initial value to 0. The method processing\_result is "dicing" a random number. Depending on the value, the quality of the part is set.



**Fig. 3.96** Example Frame

```

is
  quality: real;
do
  -- dice quality between 0 and 1
  quality:=z_uniform(?randomSeed,0,1);
  -- set the quality of the part
  -- 2 scrap
  -- 1 rework
  -- 0 o.k.
  if quality > 0 and quality
    <= self.~.scrap_rate then
      @.quality:=2;
  elseif quality > self.~.scrap_rate and
    quality <= (self.~.scrap_rate +
      self.~.rework_rate) then
      @.quality:=1;
  else
    @.quality:=0;
  end;
end;

```

After the conveyor Line, the material flow has to branch: io parts go straight to Line3, rework parts to Line2 and scrap to Line1. You can achieve this through the exit strategy of the conveyor Line. Select in the tab Exit Strategy—MU Attribute and click on Apply. Then select Attribute type—Integer and click Open List. Enter in the table in the column Attribute "quality" and, in the following columns, the value and the successor to which the MU should be moved if the attribute has the appropriate value (Fig. 3.97).

	Attribute	Value	Successor
1	quality	0	2
2		1	3
3		2	1

**Fig. 3.97** Attribute Table

# Chapter 4

## Information Flow, Controls

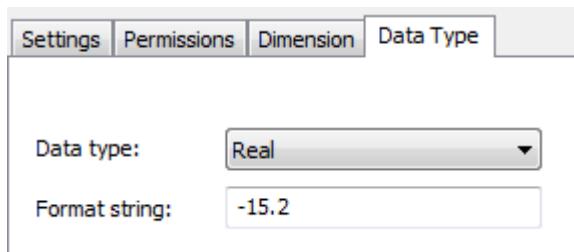
Information objects are used for managing information and data. In addition to the global variable and the method, the following objects are information flow objects:

- Lists and tables
- Trigger and generator
- AttributeExplorer
- Objects for data exchange

A large number of examples of tables and lists are available in the chapter entitled "Simulation of transport processes" as well.

### 4.1 The List Editor

Enter values and select settings in lists and tables in the list editor. Duplicate the object TableFile in the class library, and open the duplicate by double-clicking it. Each column has a data type and each cell has a unique address. To type data into a table or list in a frame, you have to turn off inheritance: Format—Inherit Format. For setting the data types of each column, it is best to use the context menu (right-click on the column header) Format (Fig. 4.1).



**Fig. 4.1** TableFile Format

Select the data type of the column here. The format string restricts interactive input (validity of entries). For example, -9 means that numbers with a maximum

of nine digits can be entered and that negative numbers are allowed; -15.2 limits an input and display to two decimal places. If you insert a list of values in the format string field (separated by semicolons, e.g. wood; metal; paper), a drop-down list is displayed when you double-click in the cell (Fig. 4.2).

	string 1	string 2
1	metal	
2	metal	▼
3	wood	
4	metal	
		paper

**Fig. 4.2** DropDown List

On the Permissions tab, you can endow cells with write protection.

- Editor: Writing in the list editor is not possible (read only).
- Information flow (read only): Write access by methods is not possible.

Distinct settings can be made for column, row and cell. In the Dimension tab, you can limit the number of displayed columns and rows. Outside of the set list dimension you cannot enter data (even through SimTalk). You can assign column and row indices, which is especially helpful for working with tables. By default, these row and column indices are hidden. You can display the indices with Format—Row Index—Active or Format—Column Index—Active. The row and column indices have an index of zero and are of the data type string.

## 4.2 One-Dimensional Lists

With CardFile, QueueFile and StackFile, Plant Simulation provides three elements for mapping one-dimensional lists. These objects are unavailable in the standard license.

### 4.2.1 *The CardFile*

The CardFile is a one-dimensional list with random access to the content via an index. You can use the CardFile like an array, storing and reading many values under one name. This object can easily store data with the same data type. When you insert entries, Plant Simulation moves the following entries back one position. You can remove (and read) entries (with “[]”). However, you can also read entries without removing the entry from the CardFile (with the command read(int index)).

**Example: Materials List**

Insert a CardFile with the name “material” and a method into an empty frame. Turn off inheritance. Enter the values from Fig. 4.3.

	string
1	wood
2	stone
3	steel
4	reed

**Fig. 4.3** CardFile

A method now is to read line 2 (define a method in the same frame). The following commands are required to do so.

```
is
do
    print material.read(2);
end;
```

Run the method (run—run or F5). The console should display “stone” at this point. “Gold” should now be inserted in line 3. For inserting entries, the CardFile provides the method insert (position, value). Change the method as follows:

```
is
do
    material.insert(3, "gold");
end;
```

Run the method with F5.

The most important attributes and methods of the CardFile are presented in Table 4.1.

### 4.2.2 *StackFile and QueueFile*

StackFile and QueueFile are one-dimensional lists that are accessed according to the FIFO (first in, first out; queue) or LIFO (last in, first out; stack) principle. New entries will be inserted into the StackFile at the top of the list; the last element inserted is the first entry to be removed. New entries will be added to the QueueFile at the bottom of the list and the first element will be removed. The main methods for working with stacks and queues are push (element) and pop. With delete, the entire content of a list will be deleted.

**Table 4.1** Attributes and Methods of the CardFile

Method/Attribute	Description
<code>&lt;path&gt;.insert(&lt;integer&gt;,&lt;value&gt;);</code>	Inserts the value <code>&lt;value&gt;</code> at the position <code>&lt;integer&gt;</code> . Entries with the same or a higher index will be moved. In QueueFile and StackFile, you only pass the value; in the QueueFile, insertion takes place at the last position, and in the StackFile at the first position.
<code>&lt;path&gt;.cutRow(&lt;integer&gt;);</code>	Removes the entry with the index <code>&lt;integer&gt;</code> , all other entries move up. The method returns the removed entry. For QueueFile and StackFile, you pass no index. In the QueueFile, the method deletes the first position, in the StackFile the last.
<code>&lt;path&gt;.read(&lt;integer&gt;);</code>	Reads the entry with the index <code>&lt;integer&gt;</code> without removing it
<code>&lt;path&gt;.append(&lt;value&gt;);</code>	Appends the passed value to the end of the list
<code>&lt;path&gt;[&lt;integer&gt;]</code>	Returns the value at the position <code>&lt;integer&gt;</code> and deletes the entry
<code>&lt;path&gt;.dim</code>	Returns the number of entries
<code>&lt;path&gt;.empty</code>	Returns true if the list contains no entries
<code>&lt;path&gt;.delete</code>	Deletes the complete content of the list

### 4.2.3 *Searching in Lists*

The process of searching is as follows: Plant Simulation works when searching with an internal cursor. This cursor is set by Plant Simulation to the position of the hit. With the position of the cursor, you can determine the position of the hit within the list. At the beginning of a search, the cursor may still be on the old "Find" position. Therefore, it is necessary to first set the cursor to position 1: `<list>.cursor: = 1;` Then, you can use the method `<list>.find(<value>)` to search for the value. The find method returns true if the value was found and false if the value was not found. The cursor is placed in the case of finding the hit position. In a third step, you need to read the cursor position: position: `<list>.cursor`.

## 4.3 The TableFile

The TableFile is a two-dimensional list, which allows random access to the entries via their address. TableFiles have many fields of application in simulation projects, e.g.:

- Storage of work plans and production orders
- Collection of statistical information
- Parameterization of models

### 4.3.1 *Methods and Attributes of the TableFile*

You can access values within the TableFile using `<table>[x,y]` (read and write). The following notation applies to ranges in TableFiles:

- One cell: `table[column, row]`
- Range: `{column1, row1}..{column2, row2}`

The range specification consists of two direct specifications, which are separated by two periods. The first specification defines the upper left corner, while the second determines the lower right corner of the range. All entries in the rectangular area will be evaluated. When you enter `{*,*}` as a second indication, the table evaluates the largest valid column and row index.

Samples (see also Table 4.2):

`{2,2}..{3,3}`  
`{"front", "door"}..{"rear", "door"}`  
`{3,1}..{*,*}`

**Table 4.2** TableFile Range Notations

Notation	Range
<code>{1,2}..{3,5}</code>	from column 1 to column 3 and row 2 to row 5
<code>{1,*}..{4,*}</code>	all rows in column 1 to column 4
<code>{2,3}..{*,3}</code>	in row 3, all columns starting with column 2
<code>{2,3}..{*,*}</code>	all the columns from column 2 and all rows from row 3
<code>{*,*}..{3,5}</code>	all columns to column 3, and all lines to line 5

Methods of the TableFile you can find in Table 4.3.

**Table 4.3** Methods of the TableFile

<code>&lt;path&gt;.delete</code>	Deletes all entries or the specified range in the table/list
<code>&lt;path&gt;.delete(&lt;range&gt;)</code>	
<code>&lt;path&gt;.copy</code>	Copies the contents of the cells
<code>&lt;path&gt;.copy(&lt;range&gt;)</code>	to the clipboard
<code>&lt;path&gt;.initialize (&lt;ranges&gt;,&lt;values&gt;)</code>	Preallocates the specified areas with the passed values, existing contents will be overwritten
<code>&lt;path&gt;[&lt;column&gt;,&lt;row&gt;]</code>	Read/write access; the TableFile allows random access via column and row indices. The index starts and ends with a square bracket. Within the brackets, you first enter the column and then the row of the cell you want to access. If you assign a value, the data type of the value must match the data type of the cell.
<code>&lt;path&gt;[&lt;column&gt;,&lt;row&gt;]:= &lt;value&gt;;</code>	
<code>&lt;path&gt;.yDim</code>	Returns the number of entries (lines)
<code>&lt;path&gt;.xDim</code>	Returns the number of columns (which contain values)

**Table 4.3 (Continued)**

<path>.dim	Returns the product of columns and rows
<path>.find(<range>, <value>)	Sets the cursor into the cell, which contains the value. You can determine the coordinates with the cursor (table.cursorX and table.cursorY). Before searching, make sure that the cursor is located in the correct position. For this, you can use <path>.setCursor(<column>, <row>).
<path>.insertRow(<value>)	Adds a new empty row at the position <value>
<path>.writeRow(<position>, <value1>,<value2> ...)	Replaces all entries in the row starting at the specified position by the passed arguments
<path>.sort(<column>, “up”/“down”)	Sorts the value of a column ascending (up) or descending (down)

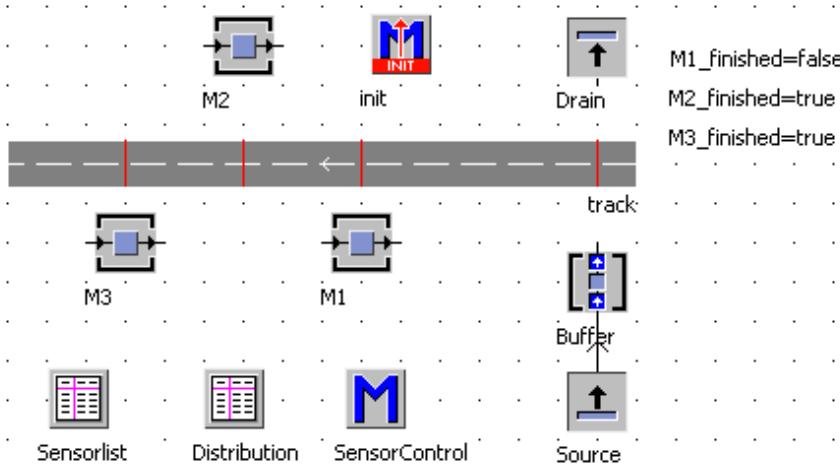
### 4.3.2 *Searching in TableFiles*

To search in tables, you must perform the following steps:

- Set the cursor in the first cell that you want to search (method setCursor(<integer x>,<integer y>)).
- Next, call the method Find. This returns false if no match was found. If a match is found, the cell cursor is moved to the corresponding cell and the method returns true.
- Finally, read the position of the cell cursor (cursorX and/or cursorY). Using the cursor position, you can access the requested cell in the TableFile.

#### **Example: Searching in TableFiles (Gantry loader)**

In a flexible manufacturing line, a loader loads three different machines. The parts to be produced have different manufacturing processes. To simplify the example, the parts are produced one after another in the line. The processing sequence is saved in a sequence list for each part (user-defined attribute of the part). The machines are managed by a machine list (sensor list). Create a frame according to Fig. 4.4.



**Fig. 4.4 Example Frame**

Insert in the class library three entities (Part1, Part2 and Part3). Next, insert into the parts a user-defined attribute AFO\_Nr (integer, initial value: 1) and an attribute workplan (data type list, list items data type string). Fill the workplans of the parts lists with the following values:

Part1	Part2	Part3
M1	M2	M3
M2	M1	M1
M3		M3

Let the source randomly produce Part1, Part2 or Part3 at intervals of three minutes. Modify the SingleProc so that you can see the end of processing on a related attribute—e.g. m1\_process\_finished (= true). For instance, in the case of M1, you can write an exit control with the following content:

```
is
do
  M1_finished:=true;
end;
```

Place the sensors along the track as in Fig. 4.5.

ID	Position	Front	Rear	L.	Path
1	1m		x		SensorControl
2	7m	x			SensorControl
3	10m	x			SensorControl
4	13m	x			SensorControl

**Fig. 4.5** Sensor Positions

Enter in the table Sensorlist the machine names on the right row/sensor position (leave the first line blank, Fig. 4.6).

	string
1	1
2	M1
3	M2
4	M3

**Fig. 4.6** Sensorlist

Insert in the transporter within the class library a user-defined attribute targetSensorID (integer; initial value: 1). In the init method, create the transporter at the position of three meters, turn to backward motion and set the process\_finished variables to false. Init method:

```
is
  car:object;
do
  -- insert car
  car:=.MUs.transporter.create(track,3);
  car.backwards:=true;
  M1_finished:=false;
  M2_finished:=false;
  M3_finished:=false;
end;
```

The sensor control may look like this: On Sensor1, the empty transporter waits until the buffer is occupied and loads one part. If the transporter is occupied on Sensor1, then the loaded part will be moved to the drain first. After loading the part from the buffer, the first entry of the work-plan of the part will be read. In the sensor list, the first machine will be searched and the target sensor for the trip of the transporter determined. After setting the target sensor and the direction, the vehicle will be set in motion. The control of the remaining sensors is similar:

The loaded transporter drives to the target sensorID, where the part will be moved onto the machine. The transporter waits until the machine has finished the processing. The finished part is then loaded. Thereafter, a new sensor will be

determined and the transporter will be set in motion. The sensor control could have the following content:

```
(SensorID : integer)
is
  machine:object;
do
  if sensorID = @.targetSensorID then
    @.stopped:=true;
    -- Buffer / Drain
    if sensorID = 1 then
      if @.occupied then
        @.cont.move(drain);
      end;
      -- wait for the next part
      waituntil buffer.occupied prio 1;
      buffer.cont.move(@);
      -- find the first machine in the sensorlist
      -- set the cursor to the start of the search
      sensorlist.setCursor(1,1);
      -- seach the next machine
      sensorList.find(@.cont.workplan.read(1));
      -- read row and set the value as targetSensorID
      @.targetSensorID:=sensorList.cursorY;
      @.backwards:=false;
      @.stopped:=false;
    else -- all other sensors
      -- load part
      machine:=str_to_obj(sensorlist[1,sensorID]);
      @.cont.move(machine);
      machine.process_finished.value:=false;
      -- wait for finish
      if machine = M1 then
        waituntil M1_finished = true prio 1;
      elseif machine = M2 then
        waituntil M2_finished = true prio 1;
      elseif machine = M3 then
        waituntil M3_finished = true prio 1;
      end;
      -- move the part to the transporter
      machine.cont.move(@);
      -- increase AFO_Nr of the part
      @.cont.AFO_Nr:=@.cont.AFO_Nr+1;
      -- find next machine, at the end to Sensor 1
      if isVoid(@.cont.workplan.read(
        @.cont.AFO_Nr+1))
      then
        @.targetSensorID:=1; -- to the drain
```

```

else
    sensorlist.setCursor(1,1);
    -- find machine
    sensorlist.find(@.cont.workplan.read
        (@.cont.AFO_Nr+1));
    -- set new destination
    @.targetSensorID:=sensorlist.zeigerY;
end;
-- set direction
@.backwards:= (sensorID > @.targetSensorID);
@.stopped:=false;
end;
end;
end;

```

### 4.3.3 Calculating within Tables

Tables and lists provide a number of methods for calculating certain values (Table 4.4).

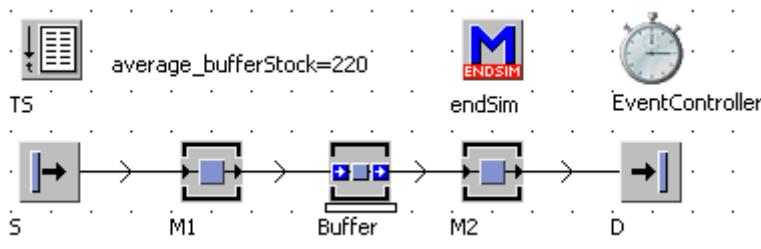
**Table 4.4** Methods for Calculating within Tables

Method	Description
<path>.max(<range>)	Provides the maximum value of the range
<path>.min(<range>)	Provides the minimum value of the range
<path>.meanValue(<range>)	Calculates the average of the values in the given range of cells
<path>.sum(<range>)	Returns the sum of the values in the range
< path >.maxAttr(<range>, <attributeName>)	If you have an object list, then e.g. maxAttr returns the maximum value of the attribute with the name <attributeName> (string) and sets the cell cursor in this row
< path >.minAttr(<range >, <attributeName>)	
< path >.meanValueAttr(<range >	
<attributeName>)	
< path >.sumAttr(<range >, <attributeName>)	

You can use these methods for all types of lists—e.g. for TimeSequence, TableFiles or QueueFiles.

#### Example: Calculate in Tables—Average Buffer Stock

In a simulation, you should record a timeline of the stock in a buffer. At the end of the simulation, calculate from it the average buffer stock. Create a simple frame as in Fig. 4.7.



**Fig. 4.7** Example Frame

Prepare the following settings: M1: processing time of two minutes; M2: processing time of one minute, 50 per cent availability, 30 minutes MTTR; buffer: capacity 1,000. The TimeSequence records buffer.numMU every minute. The method endSim should determine the average stock in the buffer based on the recorded data of the TimeSequence. Then, the data of the TimeSequence will be deleted. For this, the following programming is necessary:

```
is
do
  average_bufferStock:=TS.meanValue({2,1}..{2,*});
  --delete TimeSequence
  TS.delete;
end;
```

#### 4.3.4 Nested Tables and Nested Lists

You can select in a table for a column the data type list or table. This means that each cell in this column may contain one table or list. You can also dynamically create nested tables. To do this, first set in the table Column the format of the sub-table. Then, create with the method `<path>.createNestedList(<integer>, <integer>)` the sub-table at the designated x, y position.

##### Example: Delivery Calls

The following example shows how to create a delivery call list for a warehouse simulation. The calls each consist of one to seven different products with a quantity between 10 and 100 units per product. This should generate 10,000 calls. Create a simple frame as in Fig. 4.8.

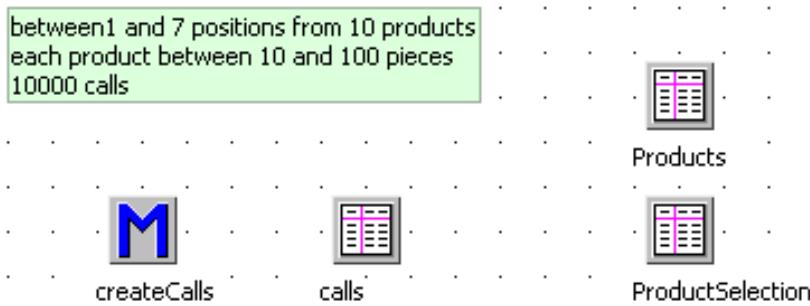


Fig. 4.8 Example Frame

The table Products contains all products available for selection (string). Enter in the first column 10 product names (e.g. P1 to P10). The table ProductSelection you can retain in the default formatting. The table Calls has the data type table in the first column. The sub-tables contain in the first column the product names and in the second column the number of this product in the call (see format in Fig. 4.9).

	string	integer	st
string	Product	Quantity	
1			
2			
3			

Fig. 4.9 Sub-table Format Table Calls

The generation of each call could be carried out as follows:

- Create for the call a new sub-table
- For the selection of products, copy all products into the table ProductSelection
- Dice the number of products
- The product is "drawn" from the table ProductSelection (remove the product from the table)
- Dice the quantity of the product
- Write product and quantity in a row of the sub-table

The method createCalls could have the following content:

```

is
  i,k:integer;
  number:integer;
  prodNo:integer;
  product:string;
  prodNumber:integer;
do
  --delete all old entries
  calls.delete;

```

```
for i:=1 to 10000 loop
  --create new subtable
  calls.createNestedList(1,calls.yDim+1);
  -- dice number of products
  number:=round(z_uniform(1,1,7));
  productSelection.delete;
  --copy products to product-selection
  products.copyRangeTo({1,1}..{1,*},
    productSelection,1,1);
  for k:=1 to number loop
    --dice one product würfeln
    prodNo:=round(z_uniform(2,1,
      productSelection.yDim));
    product:=productSelection[1,prodNo];
    --delete row from productSelection
    productSelection.cutRow(prodNo);
    --dice quantity
    prodNumber:=round(z_uniform(3,10,100));
    --write row in the nested table
    calls[1,calls.yDim].writeRow(1,
      calls[1,calls.yDim].yDim+1,
      product,prodNumber);
  next;
  next;
end;
```

## 4.4 TimeSequence

The TimeSequence is part of the professional license of Plant Simulation and is unavailable in the standard license.

### 4.4.1 *The Object TimeSequence*

You can use the TimeSequence for recording and managing temporary value progressions (stocks, machine output). The TimeSequence has two columns: point in time (first column) and value (second column). You can enter values into the TimeSequence with SimTalk, or the TimeSequence can record values by itself. The values of the TimeSequence can be analyzed relatively easily using the chart object.

The tab Contents shows the recorded values. You can sort the values in ascending order according to time. The button Set Values sets empty fields to a default value. On this tab, you must specify the data types to be stored. This is analogous to the tables:

1. Turn off inheritance (Format—Inherit Format).
2. Right-click on the column header of the second column and select Format from the menu.
3. Select the data type and click OK.

In the field Time reference within the tab Start Values, you can specify whether Plant Simulation shows time-related data in absolute format (date-time) or in relative format (time). Enter in the field Reference time the start of the recording of the values (time, date). The time values are shown relative to this reference value (which shifts the values of the time axis).

In the tab Record you can select the settings required for collecting the data. Value: Enter here the relative or absolute path to the value (method, variable, attribute), whose course the TimeSequence will record over time. You might, for example, record the number of parts in the object buffer. The method is buffer.numMU. You can select the value using the button next to the input field methods, attributes and variables. Mode: Watch means that values are entered after each change in value. This may possibly lead to a slowdown of your simulation. Sample means that values at certain time intervals are entered (e.g. every 30 minutes). In the watch mode, only observable values will be recorded. Active: Use this to activate or deactivate the TimeSequence.

### Example: TimeSequence

A process must be balanced. Three machines supply a fourth machine with parts. The machines M1, M2, M3 have very low availabilities (time-consuming tool testing and adjustments). You are looking for the maximum output of the line, the processing time of M4 and the required buffer size. Create a frame as in Fig. 4.10.

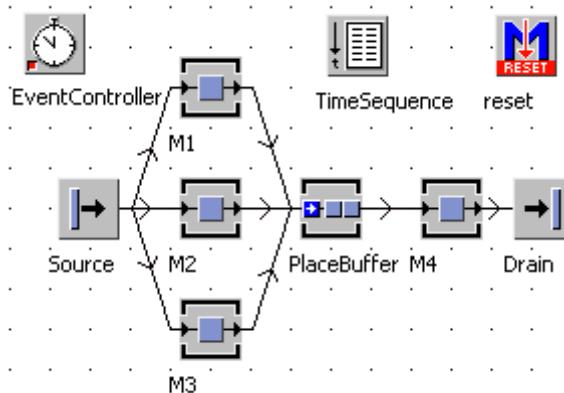


Fig. 4.10 Example Frame

Settings: Source interval of 50 seconds, blocking; M1, M2, M3 processing time of one minute, 50 per cent availability, 45 minutes MTTR; PlaceBuffer capacity 10,000 parts, zero seconds processing time; M4: 40 seconds of processing time, 75 percent availability, 25 minutes MTTR. The course of stock in the PlaceBuffer is to be recorded in the TimeSequence.

Follow these steps:

1. Turn off inheritance: Format—Inherit Format (remove the check mark, Fig. 4.11).

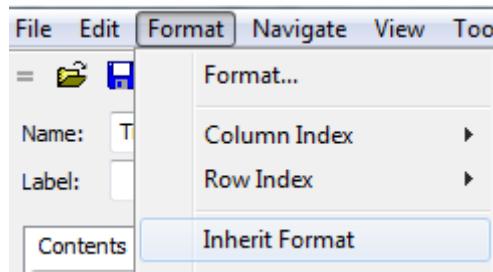


Fig. 4.11 TimeSequence—Inherit Format

2. Click the tab Record, and replicate the settings from Fig. 4.12.

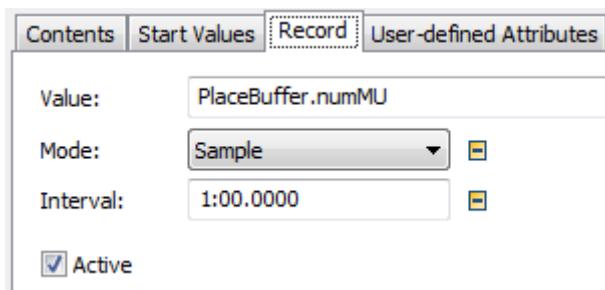


Fig. 4.12 TimeSequence settings

3. Start the simulation. The current time of the EventController and the stock in the PlaceBuffer will be entered into the TimeSequence every minute.

You can easily export the values of the TimeSequence (e.g. as a text file). First, select the format of the text file: File—Format. Save the table with File—Save as text.

Note: The EventController does not reset the TimeSequence. You must delete the previous content of the TimeSequence inside a reset or init method.

Example of a reset method:

```
is
do
  timeSequence.delete;
end;
```

The methods and attributes of TimeSequence are those of the TableFile.

#### 4.4.2 *TimeSequence with TableFile and SimTalk*

Unfortunately, the TimeSequence is not available in the standard license. Based on the table, you can create your own TimeSequence with a little effort. The table must have two columns (first column: time; second column: data type of the recorded attribute). In addition, you need an attribute to store the interval of the recording and one to activate the recording. You need to store an object reference and the name of the attribute. You will also need two methods (init and record). Duplicate in the class library a TableFile (name it Timeline). Insert in this table user-defined attributes as shown in Fig. 4.13.

Name	Value	Type	C.	I..	3D
active	false	boolean	*		
attribute		string	*		
init		method	*		
interval	1:00.0000	time	*		
obj	(?)	object	*		
record		method	*		

**Fig. 4.13** User-defined Attributes

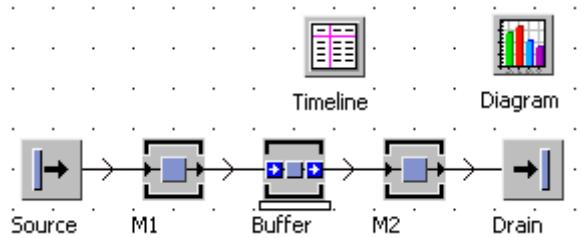
The init method first deletes all data from the table and starts, if active = true, the method Record:

```
is
do
  self.~.delete;
  if self.~.active then
    self.~.record;
  end;
end;
```

The method record writes the current time of the event controller and the value of the attribute of the object in the table and calls itself after the elapse of the interval.

```
is
do
  self.~.writeRow(1, self.~.yDim+1,
    root.eventcontroller.simTime,
    self.~.obj.getAttribute(self.~.attribute));
  self.methcall(self.~.interval);
end;
```

To test the timeline, create a simple network according to Fig. 4.14 with the table Timeline.



**Fig. 4.14** Example Frame

Settings: source interval of two minutes; M1: processing time of two minutes, buffer capacity of 1,000; M2: one minute processing time, 50 per cent availability, one hour MTTR. Set the user-defined attributes of the Timeline table as shown in Fig. 4.15 Settings.

Name	Value	Type	C.	I..	?
active	true	boolean			
attribute	numMu	string			
init		method	*		
interval	1:00.0000	time	*		
obj	Buffer	object			
record		method	*		

**Fig. 4.15** Settings

The table now collects data such as the Plant Simulation object TimeSequence.

## 4.5 The Trigger

The trigger is not part of the standard-license of Plant Simulation.

### 4.5.1 *The Object Trigger*

The trigger can change values of attributes and global variables during the simulation according to a defined pattern and perform method calls. In addition, the trigger can control a source, so that this starts to produce MUs from a certain moment in time.

#### Example: Trigger

A machining center produces parts in three shifts (24 h) with a processing time of one minute. The following assembly produces one shift with three parallel places, and two shifts with one place. The assembly time is 1:40 minutes. The parts not yet assembled are collected in a buffer. Create a frame according to Fig. 4.16.

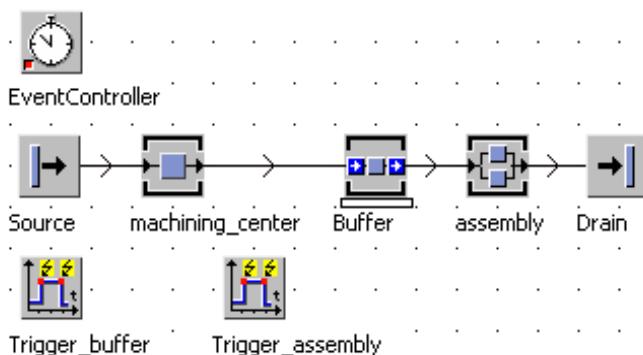


Fig. 4.16 Example Frame

Settings: Source one minute interval, blocking; machining\_center one minute processing time; buffer zero seconds processing time, capacity 1000 parts. After eight hours of simulation time, the property assembly.XDim is set to one. After additional 16 hours it must be changed to three. Select settings from Fig. 4.17 in the object Trigger\_assembly.

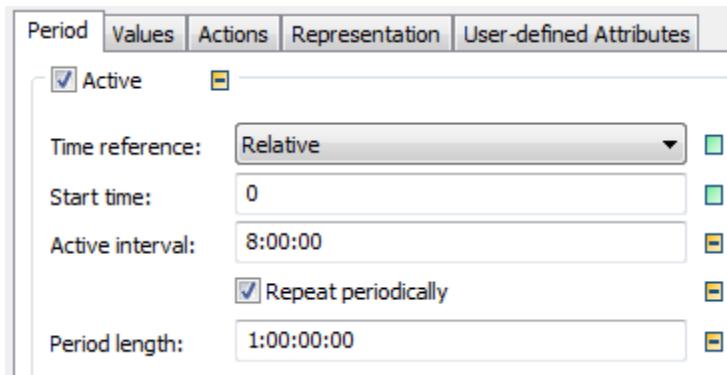


Fig. 4.17 Trigger Settings

Select values, open the table by clicking on Values and enter the data from Fig. 4.18.

string	Point in Time	Value
1	0.0000	3
2	1:00:00:00.0000	3

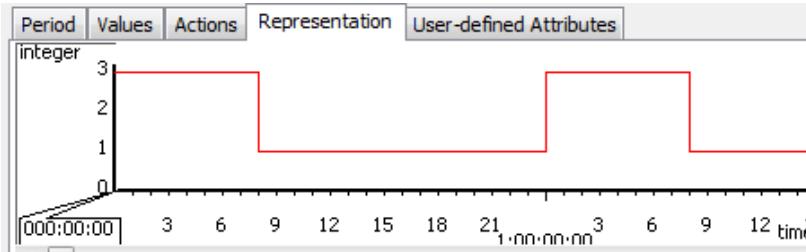
Fig. 4.18 Trigger Values

Set the default value (here one) on the tab Start values. Click on the tab Actions. Deactivate the inheritance and click on the button Attributes. Insert the data from Fig. 4.19.

	Object	Attribute	Error Message
1	.Models.Trigger.Assembly	xDim	Attribute xDim could not be set.

**Fig. 4.19** Trigger Action

You can check the distribution you set on the tab Representation (Fig. 4.20).



**Fig. 4.20** Trigger Representation

If you run the simulation for a while, you will get an error message. Plant Simulation cannot reduce the dimension of the parallel station when parts are located on the respective places. Prior to the reduction of the capacity, the object assembly needs to be emptied. This can, for example, be achieved by temporarily locking the exit of the buffer (e.g. two minutes before shift change). The following settings are needed in the object Trigger\_buffer: actions: Attribute buffer.exitLocked; start time: 7:58:00; active interval 2:00; period length: one day, repeat periodically; data type: Boolean; values: 7:58:00 true; default value false

#### Trigger parameters:

**Active:** Select whether the trigger is active or not during the simulation run.

**Time reference:** You can select a relative start time (0:00) or an absolute time (date).

**Start time:** When should the trigger be active for the first time?

**Active interval:** After what period should the value be set back to the default value (defined in a timeline of e.g. eight hours)?

**Repeat periodically:** The trigger is active again after the expiration of the period length.

**Period length:** Sets the duration of a trigger period (e.g. one day or 1:00:00:00.0000).

**Attributes:** Type into the list the attributes you want to control. An error message appears in the console when Plant Simulation cannot execute the action. Before you can type values into the table, turn off inheritance and click Apply.

**Values:** Enter the progress of the value, which the trigger controls, into a TimeSequence.

### 4.5.2 Trigger with SimTalk and TableFile

The trigger is not part of the standard license. However, you can create your own trigger using some lines of SimTalk and a TableFile. You need a TableFile in which changes in the value are recorded within a certain time sequence. This table is evaluated by a method and the corresponding values of the object will be set. You can program the entire trigger within a TableFile.

#### Example: Programming of a Trigger Using SimTalk

You need to model the changing power consumption of machines during the processing time in Plant Simulation. For this purpose, a table is defined for the distribution of the power consumption during the processing for each machine. The trigger has the function to set the current power consumption of the machine (so you can read it from the outside). The power consumption of the entire system is checked and recorded through another TableFile. Duplicate a SingleProc and rename the duplicate as "machine." Add user-defined attributes according to Fig. 4.21 into the machine (in the class library).

Name	Value	Type
act_consum	5	real
base_consum	5	real

Fig. 4.21 User-defined Attributes

Create two duplicates of the TableFile in the class library (Trigger\_ and power\_consumption, both tables containing in the first column the data type time and in the second column real). We will develop both elements on the basis of the block TableFile. Both tables should contain only two columns. The number of columns in the tables is set with the maxXDim property. If you do not want to set this property with SimTalk, then you can do this by using the context menu and the "Show attributes and methods" dialog. Double-click on the property maxXDim and set two as the value (Fig. 4.22).

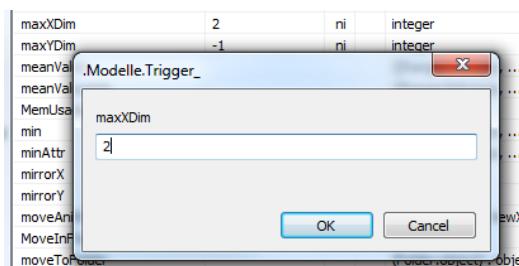


Fig. 4.22 Attribute maxXDim

Create user-defined attributes according to Fig. 4.23 in the object Trigger\_.

Name	Value	Type	C.	I..
active	false	boolean	*	
Attribute		string	*	
init		method	*	
obj	(?)	object	*	
repeat_	false	boolean	*	
repeat_interval	0.0000	time	*	
setValue		method	*	
trigger_pos	0	integer	*	
triggercontrol		method	*	

Fig. 4.23 User-defined Attributes TableFile Trigger\_

The attributes will be used as follows:

Attribute/Method	Use
active	Activate/deactivate the trigger
attribute	The attribute we will set (must be inserted as a string)
init	Method for initializing
obj	The object whose attribute is to be set (input)
repeat_	Attribute sets whether the trigger should run repeatedly through the table of values
repeat_interval	To determine the time interval for a repeated call (reboot) of the trigger
setValue	Method is invoked by the trigger control, and sets the values in the object attribute
trigger_pos	Internal attribute for storing the position in the value table
triggercontrol	Called when the trigger is activated and it starts the method setValue

Begin with a preparatory work: If active changes from false to true, the triggercontrol should be invoked. For this to work safely, you need an initialization of active (false). Similarly, the Trigger\_pos must be reset to zero. Trigger\_.Init method:

```
is
do
  self.~.trigger_pos:=0;
  self.~.active:=false;
end;
```

The method triggercontrol should be called by an observer in the object Trigger\_. Select the following in the object Trigger\_: Tools—Select Observer—Add. Observe the internal attribute active and start the method triggercontrol (Fig. 4.24).

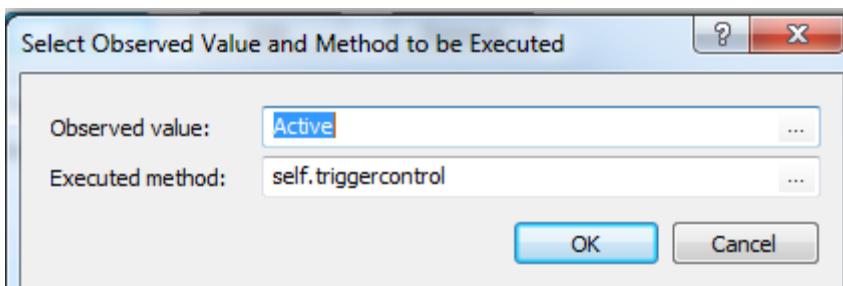


Fig. 4.24 Observer Trigger\_

The triggercontrol starts the setValue method after the first delay (the first entry in the table). Triggercontrol:

```
(Attribute: string; oldValue: any)
is
do
  if self.~.active then
    if self.~.trigger_pos = 0 then
      --call setValue after first interval
      self.~.setValue.methcall((self.~)[1,1]);
    end;
  end;
end;
```

The method setValue sets the attribute of the object to the next value and calls itself after the right time:

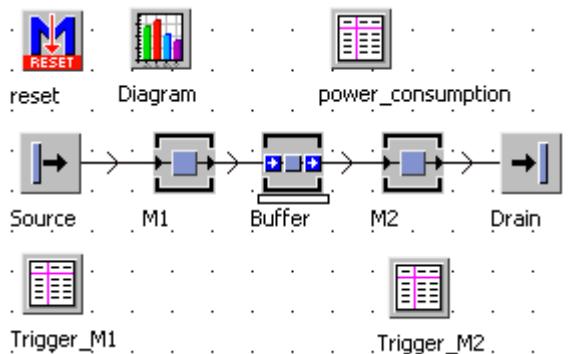
```
is
  interval:time;
do
  self.~.trigger_pos:=self.~.trigger_pos +1;
  -- set attribute to value
  self.~.obj.setAttribute(self.~.attribute,
    (self.~)[2,self.~.trigger_pos]);
  -- if last position and not repeat_
  -- reset trigger_pos and finish
  if self.~.trigger_pos = self.~.yDim then
    self.~.trigger_pos:=0;
    if self.~.repeat_ = false then
      self.~.active:=false;
      return;
    else
```

```

interval:=self.~.repeat_interval-
  (self.~)[1,self.~.trigger_pos]+(self.~)[1,1];
  self.methAufr(interval);
end;
else
  --recall after next distance
  interval:=(self.~)[1,self.~.trigger_pos +1]-
  (self.~)[1,self.~.trigger_pos];
  self.methAufr((self.~)[1,self.~.trigger_pos +1]);
end;
end;

```

Create a frame according to Fig. 4.25.



**Fig. 4.25** Example Frame

Settings: source interval of one minute; Machine1 processing time of one minute; Machine2 processing time of 45 seconds; enter an entrance control front (internal method) in Machine1 and Machine2. Activate in the entrance control the trigger\_objects for the machines—e.g. for Machine1:

```

is
do
  trigger_M1.active:=true;
end;

```

The entrance control of Machine2 is structured analogously.

Method reset:

```

is
do
  M1.act_consum:=M1.base_consum;
  M2.act_consum:=M2.base_consum;
end;

```

Settings in the triggers: Set in Trigger\_machine1, Machine1 as obj and enter in attribute act\_consum (at trigger\_machine2 according to Machine2). Insert data in the trigger table as in Fig. 4.26.

	time 1	real 2
string	Time	Value
1	2.0000	20.00
2	5.0000	30.00
3	15.0000	50.00
4	45.0000	10.00
5		

	time 1	real 2
string	Time	Value
1	5.0000	15.00
2	10.0000	55.00
3	44.0000	10.00
4		
5		

Fig. 4.26 Trigger Values

The entrance control activates the trigger for each part. The triggercontrol processes the values from the table and then completes its work (repeat = false). The trigger sets the different power values needed during processing. This happens for Machine1 and Machine2 independently, depending on when a part is machined on the machine. Next, we need a recording possibility. As the standard license has no TimeSequence, we will develop a small solution here. Place user-defined attributes in the class power\_consumption within the class library as shown in Fig. 4.27.

Name	Value	Type	C.
active	false	boolean	*
init		method	*
interval	1.0000	time	*
Machines		table	*
record		method	*

Fig. 4.27 User-defined Attributes

The attributes have the following uses:

Attribute/Method	Description
active	Activate/deactivate the recording
init	Initializing
interval	Interval between the recordings
Machines	Table of the machines that are to be taken into account (first column formatted as data type object)
record	Writes the sum of the power consumption and the simulation time in the table

Init: The table must be emptied before each run. If the active is true, the method Record will be called:

```
is
do
  self.~.delete;
  if self.~.active then
    self.~.record;
  end;
end;
```

The method Record sums all values of the act\_consum attributes of the machines of the table Machines and enters the sum and the simulation time into the table. The method calls itself once again after the interval.

```
is
  i:integer;
  power:real;
do
  for i:=1 to self.~.machines.yDim loop
    power:=power+ self.~.machines[1,i].act_consum;
  next;
  --write new row
  self.~.writeRow(1,self.~.yDim+1,
    root.ereignisverwalter.simTime,power);
  --call next
  self.methcall(self.~.interval);
end;
```

You can evaluate the power consumption using a chart component. Drag the table power\_consumption onto a chart object. Set in the register Display: XY graph; chart type: line; data: in column (Fig. 4.28).

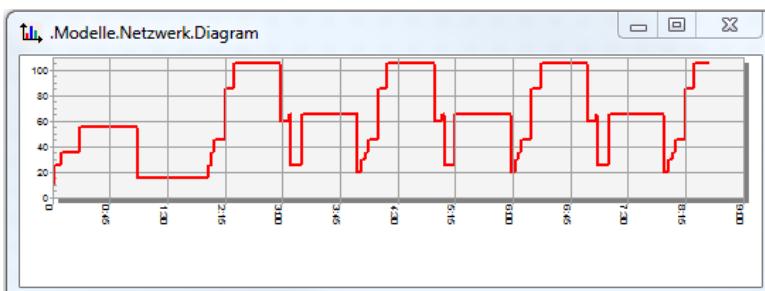


Fig. 4.28 Diagram

## 4.6 The Generator

The generator is not part of the standard license. But analogously to the TimeSequence and the Trigger, you can create your own generator from the available blocks.

### 4.6.1 The Generator Object

The generator starts a method at regular intervals or after a certain time has passed. You can specify all times as a fixed time or as a statistical distribution.

#### Example: Generator, Outward Stock Movement

In the following frame, the produced parts will not be removed by a drain; instead, they will be placed in a store (capacity: 10,000). To ensure that the store does not overflow after a short time, we need to simulate outward stock movement. The store will have an average outward stock movement of 80 units per hour. For this purpose, you need a method that removes 80 parts per hour from the store. Create a frame like in Fig. 4.29.

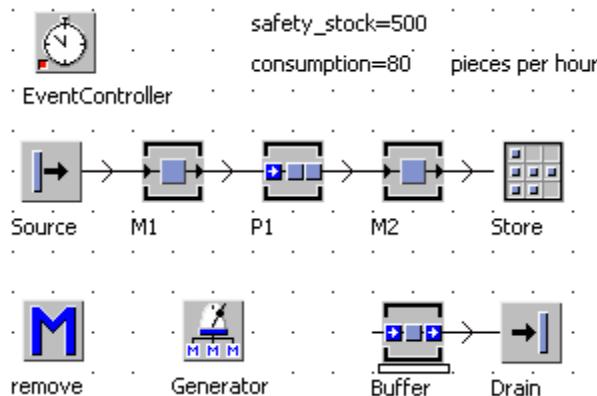


Fig. 4.29 Example Frame

Settings: Source interval: one minute blocking; M1 and M2 processing time: 50 seconds; availability: 95 per cent; MTTR: five hours, P1 capacity: 1,000 parts; store capacity: 10,000 parts. Create safety\_stock and outward stock movement (consumption) as global variables, of the data type integer in the frame. Program the method Remove (called once an hour) as follows:

```

is
  i:integer;
do
  if store.numMU >= (consumption+safety_stock)
    then

```

```
--remove MUs
for i:=1 to consumption loop
  store.cont.move(buffer);
next;
end;
end;
```

In the example above, the method must be called hourly. Use the generator to determine the time and the method that should be called (Fig. 4.30).

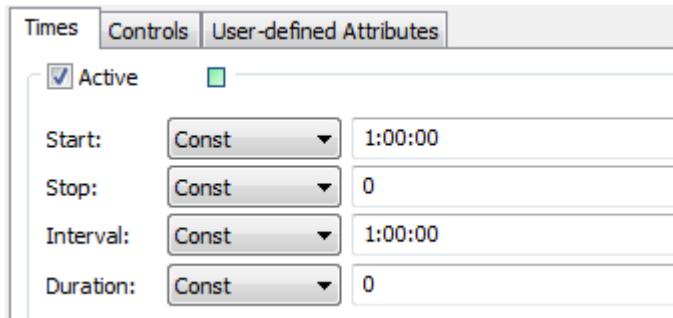


Fig. 4.30 Generator, Tab Times

In the tab Controls, select the method remove for the interval (Fig. 4.31).

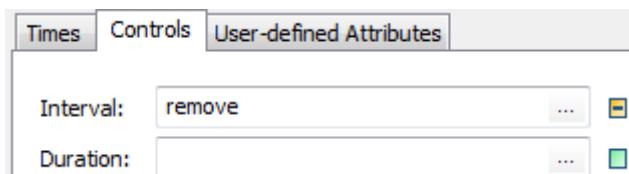


Fig. 4.31 Generator, Controls

The following settings must be made in the generator:

**Active:** Activate the generator.

**Start:** Select when the interval control will be activated for the first time.

**Stop:** Select at which simulation time no interval control should be active.

**Interval:** What time should elapse between calls?

Select your method in the tab Controls.

## 4.6.2 *User-defined Generator with SimTalk*

Unfortunately, the generator is not part of the standard license. But by using a few lines of SimTalk, you can create your own generator that calls a method (e.g. for recording statistical data) at regular intervals. To create a generator, you need the following components:

- A block as information source
- Internal methods: init to start, a method for periodically calling the control
- Variables for the interval (time) and the start delay (time)
- Possible variables for input parameters

### Example: Generator Using SimTalk

You are requested to create a generator based on a SingleProc. Through the generator, parts should be moved out of a warehouse. Duplicate a SingleProc (in the folder networks). Create a frame according to Fig. 4.32.

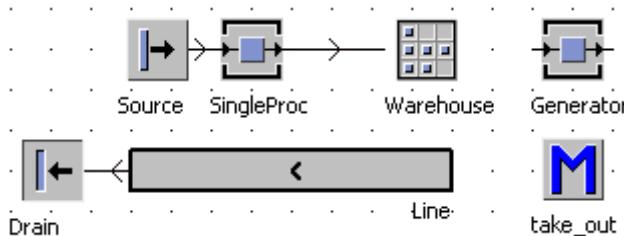


Fig. 4.32 Example Frame

Insert user-defined attributes in the SingleProc duplicate as in Fig. 4.33.

Importer		Failure Importer		User-defined Attributes						
				New		Edit		Delete		
Name	Value	Type	C.	I..	3D					
control	(?)	object	*							
init		method	*							
internMethod		method	*							
interval	0.0000	time	*							
start	0.0000	time	*							

Fig. 4.33 User-defined attributes

The init method calls after start (time) the method internMethod:

```
is
do
  self.~.internMethod.methcall(self.~.start);
end;
```

The method internMethod calls the method Control and after the interval (time) calls itself.

```

is
do
  -- call control
  ref(self.~.control).methCall(0);
  -- after interval --> next call
  self.methcall(self.~.interval);
end;

```

The method `take_out` moves one part from the warehouse to the line (if the warehouse is not empty).

```

is
do
  if warehouse.empty = false then
    warehouse.cont.move(line);
  end;
end;

```

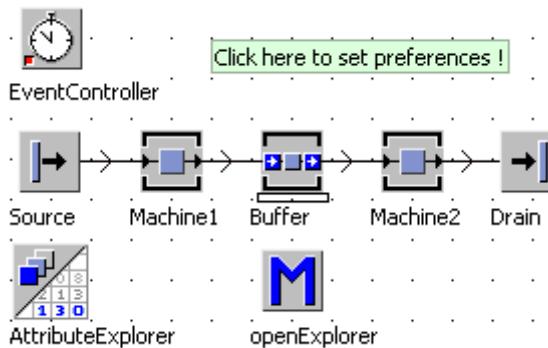
To move one part per minute starting after an hour with the help of the generator, you must modify the settings as in Fig. 4.34 within the generator.

Name	Value	Type	C.	I
control	take_out	object		
init		method		
internMethod		method		
interval	1:00.0000	time		
start	1:00:00.0000	time		

**Fig. 4.34** Generator Settings

## 4.7 The AttributeExplorer

You can manage a variety of attributes of different objects from a single central location using the AttributeExplorer. Create a frame as in Fig. 4.35.



**Fig. 4.35** Example Frame

Insert a comment with the text "Click here to set preferences!" (to open the AttributeExplorer). The processing times of Machine1 and Machine2 should be changed in a single dialog box. Add an AttributeExplorer to the frame. Open the AttributeExplorer by double-clicking it. Select the tab Objects and turn off inheritance (+ Apply). Drag the items from the frame to the list of object paths. Press Enter after each object to add a new line to the table (Fig. 4.36).

Data	Objects	Attributes	Query	User-defined Attributes
Objects				
1	.Models.AttributeExplorer.Source			
2	.Models.AttributeExplorer.Machine1			
3	.Models.AttributeExplorer.Buffer			
4	.Models.AttributeExplorer.Machine2			

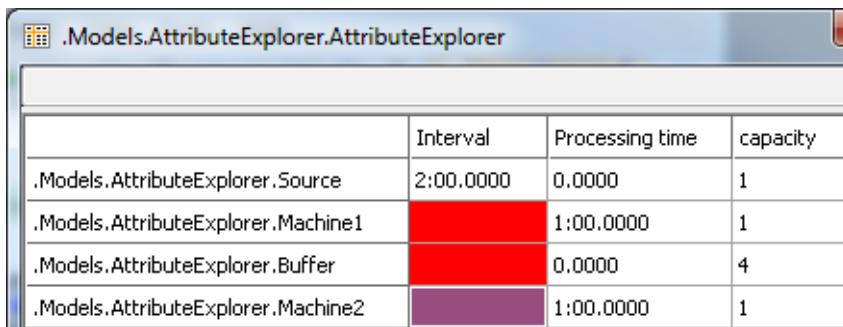
Fig. 4.36 AttributeExplorer: Object List

Next, select the attributes that you want to view and modify. In the example above, these are the attributes interval (source), procTime (Machine1 and Machine2) and capacity (buffer). Enter these settings on the tab Attributes. Use the column alias to display a name other than the attribute name in the AttributeExplorer (for instance, processing time instead of procTime). You can select the attribute using the button Show attributes. First, turn off inheritance and click Apply (Fig. 4.37).

Data	Objects	Attributes	Query	User-defined Attributes
Attributes				
1	Interval		Alias	
2	ProcTime		Processing time	
3	capacity			

Fig. 4.37 AttributeExplorer Attributes

To display the alias names, select the option Show attributes with alias on the tab Data. You can also select showing the paths or the labels of the objects. Click the button Show Explorer to open a window in which you can set all the values at once (Fig. 4.38).

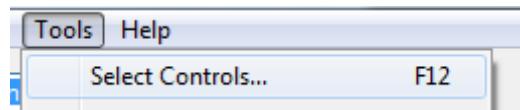


The screenshot shows a software window titled ".Models.AttributeExplorer.AttributeExplorer". Inside, there is a table with four columns: "Interval", "Processing time", and "capacity", along with a header row. The data rows are as follows:

	Interval	Processing time	capacity
.Models.AttributeExplorer.Source	2:00.0000	0.0000	1
.Models.AttributeExplorer.Machine1		1:00.0000	1
.Models.AttributeExplorer.Buffer		0.0000	4
.Models.AttributeExplorer.Machine2		1:00.0000	1

**Fig. 4.38** AttributeExplorer—Window

The AttributeExplorer should open when you click the comment ("Click here..."). Open the comment. Select Tools—Select Controls (Fig. 4.39).



**Fig. 4.39** Select Controls

Enter in the field Select the method openExplorer. Program the method openExplorer:

```
is
do
  -- activate the AttributeExplorer
  attributeExplorer.Active:=true;
end;
```

When you now click on the comment, the AttributeExplorer opens. An issue results in this way in older versions of Plant Simulation. Then, you cannot open the dialog of the comment by double-clicking (before Version 9). You can open the dialog via the structure of the frame. Right-click on the frame in the class library. Select Show structure from the menu. You can then double-click the objects in the opening window and, thus, open their dialogs. Another way (starting with version 8.2) is right-clicking on Open.

## 4.8 The EventController

The EventController enables access to the simulation time and you can control the simulation. Furthermore, you can call all buttons of the EventController in SimTalk (Table 4.5).

**Table 4.5** SimTalk attributes Attributes and methods Methods of the EventController

Method/Attribute	Description
<path>.SimTime	Returns the current simulation time (data type time)
<path>.AbsSimTime	Returns the current simulation time (data type datetime)
<path>.start	Controls the experiment runs
<path>.step	
<path>.stop	
<path>.reset	
<path>.startStat	Sets/reads the time when the EventController resets the statistics of the model (time)
<path>.isRunning	Returns true if the simulation is running; you can observe this attribute and start. e.g. methods, when the simulation is started or stopped

Plant Simulation provides a number of methods to evaluate absolute time information and filter out certain information (Table 4.6).

**Table 4.6** SimTalk Methods for Evaluating DateTime Values

Method	Description
day(<dateTime>)	Returns the day of the specified date
dayOfWeek(<dateTime>)	Returns the number of days since the last Sunday (integer, e.g. 1 for Monday)
getDate(<dateTime>)	Number of days since January 1 of the year (integer)
week(<datetime>)	Week (integer)
calendarWeek(<datetime>)	
year(<datetime>)	Number of years since 1900 (integer)
timeOfDay(<dateTime>)	Returns the time portion of the DateTime value (time)

### Example: Drive by Schedule

In the following example, a transporter drives as per a schedule with a number of stops. The schedule contains the departure times at individual stops on certain weekdays. Set a scaling factor of 50 in the frame. Create a frame according to Fig. 4.40.

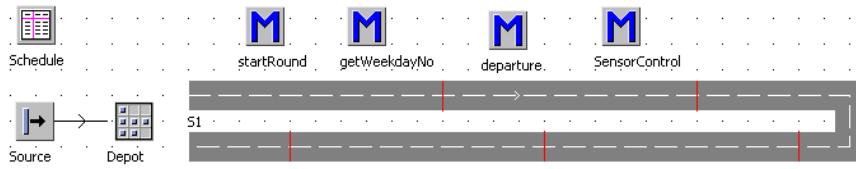


Fig. 4.40 Example Frame

Each 10 km, insert on S1 a sensor and assign to each sensor the method SensorControl. The entrance control of the depot is the method startRound. Add to the track S1 an exit control. The source generates exactly one transporter. Enter the data according to Fig. 4.41 in the table Schedule.

	string 1	string 2	string 3	string 4	st 5
string	Stop	Departure	from	to	
1	0	06:30	monday	saturday	
2	1	07:00			
3	2	07:30			
4	3	08:00			
5	4	08:30			
6	5	09:00			
7	1	09:30			
8	2	10:00			
9	3	10:30			
10	4	11:00			
11	5	11:30			
...					

Fig. 4.41 Schedule

Insert in the table Schedule a user-defined attribute schedulePos (integer). It should save the position of the transporter within the schedule. In the depot, the transporter must wait until the first trip starts. The start time depends on the time (schedule[2,1]) and the weekday. Set the EventController so that the simulation runs begin on a Sunday. For better handling of the weekdays, we will create a little helper method, which returns a number for each day of the week (starting with zero for Sunday to six for Saturday). The method getWeekdayNo could have the following content:

```
(wd:string):integer
is
  weekday:integer;
do
```

```

inspect wd
  when "sunday" then weekday:=0;
  when "monday" then weekday:=1;
  when "tuesday" then weekday:=2;
  when "wednesday" then weekday:=3;
  when "thursday" then weekday:=4;
  when "friday" then weekday:=5;
  when "saturday" then weekday:=6;
end;
return weekday;
end;

```

First, the method startRound must determine the next date on which the transporter must drive. In the second step, a starting time must be calculated based on the date of the next day and the time from the schedule. The transporter must then wait for a defined period before moving to S1. This could be implemented as follows:

```

is
  weekday:integer;
  _time:datetime;
  found:boolean;
  i,_from,_to:integer;
  startingTime:string;
  _z:datetime;
do
  _time:=eventController.absSimTime;
  weekday:=DayOfWeek(_time);
  _from:=getWeekdayNo(schedule[3,1]);
  _to:=getWeekdayNo(schedule[4,1]);
  --look for the next day in the schedule
  for i:=1 to 7 loop
    weekday:=weekday+1;
    if weekday=6 then
      weekday:=0;
    end;
    --a day in the schedule
    if weekday>=_from and weekday <= _to then
      exitLoop;
    end;
  next;
  --create a dateTime-value
  startingTime:=
    to_str(getDate(eventController.absSimTime+
      (i*24*3600)))+" "+schedule[2,1]+":00.00";
  --move the car at the startingTime
  _z:=str_to_datetime(startingTime);
  wait(_z-eventController.absSimTime);
  @.move(s1);

```

```

schedule.schedulePos:=2;--new round
end;

```

The sensors on the track represent the stops of the transporter. At the stops, the transporter stops. Thereafter, the planned departure time is determined from the schedule. At this time, the method `departure` starts the transporter and increments the position within the schedule. The `SensorControl` could have the following content:

```

(SensorID : integer; isRear : boolean)
is
  startingTimeTxt:string;
  startingTime:DateTime;
do
  @.stopped:=true;
  --calculate starting time from schedule
  startingTimeTxt:=
    to_str(getDate(eventController.absSimTime))+"
    "+schedule[2,schedule.schedulePos]+":00.0000";
  startingTime:=str_to_datetime(startingTimeTxt);
  --departure
  ref(departure).methCall(startingTime,@);
end;

```

Method `departure`:

```

(car:object)
is
do
  car.stopped:=false;
  schedule.schedulePos:=schedule.schedulePos+1;
end;

```

At the end of the track, we check whether the schedule contains additional entries (then the transporter will be moved to the beginning of the track) or not (then the transporter will be moved to the depot). The exit control of the track has the following content:

```

is
do
  if schedule.schedulePos>schedule.yDim then
    @.move(depot);
  else
    @.move(?);
  end;
end;

```

With the function `sysDate`, you can access the system date of the computer on which the simulation is running. You may use this, for example, to determine the duration of the execution of a simulation run.

### Example: Calculate Duration of Experiment Run

Create a simple frame according to Fig. 4.42.

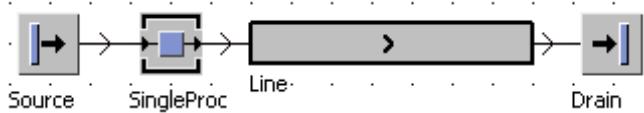


Fig. 4.42 Example Frame

Set in the EventController a simulation period of 1,000 days. Insert in the EventController a user-defined attribute sysStart (datetime). To detect when the simulation has been started and terminated, use the attribute isRunning. Select in the EventController Tools—Select Observer—Add. Select at observed value IsRunning, place the cursor in the field Executed method and press the F4 key. Confirm with OK. If the attribute changes from false to true (the simulation has begun), we store the start time in the variable sysStart. If the attribute changes from true to false, then we calculate the simulation time and output them to the console. The observer method could appear as follows:

```

(Attribut: string; oldValue: any)
is
do
  if oldValue=false then
    --simulation started
    self.~.sysStart:=sysDate;
  else
    --simulation stopped
    --write duration to console
    print "Duration of simulation: " +
      to_str(sysDate-self.~.sysStart);
  end;
end;
  
```

## 4.9 Shop Floor Control, Push Control

If you want to simulate complex manufacturing systems, the modeling of the production control system is often necessary to generate a realistic model of the production. For the simulation, usually only these aspects are interesting that have a significant influence on material flow behavior. Problematic are studies that intended to examine whether an increase in output can be achieved with changes within the production control. Another aspect may be an intended coupling of a real production control with the simulation model. The following sections deal with various aspects of production planning and control.

### Job Shop Manufacturing

The production is organized here by manufacturing technologies. Machinery and equipment that perform similar operations are grouped organizationally and physically in a workshop. Examples include workshops for turning or stamping workpieces. In addition, stocks of raw materials and finished products are needed.

The manufacturing is done in the order listed in the work plan wherein the workpieces must be transported per job from workshop to workshop. Due to the lack of coordination in the processing and transport times, there will be waiting in front of the machines and high stock levels. A special challenge in the simulation of workshop systems is the lack of a fixed chain of production units. Parts run through the different workshops according to different plans; usually, the machines on which the individual processing steps can be performed are clearly defined.

To simulate a workshop production, you require the following elements:

- Jobs
- Parts and work plans
- Machines with associated queues

### 4.9.1 Base Model Machine

The following scenario must be simulated:

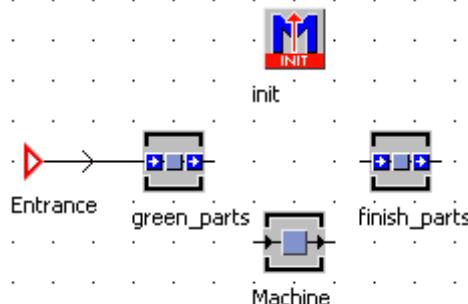
A company has mechanical manufacturing consisting of:

- four turning machines
- two milling machines
- two gear milling machines
- one induction hardening machine
- five grinding machines

There is a wide variety of manufactured shaft-like workpieces. With the help of the simulation, the expansion of production diversity should be examined and any necessary investment decisions should be secured. The transport is done manually and is to be represented only by a mean transport time. The parts to be transported are carried in containers. Each machine has a container for rough and machined parts. For easier modeling, it makes sense to model the machine as a sub-frame with the following content:

- SingleProc
- Buffer for rough parts (or green parts)
- Buffer for finished parts
- Interface (with connector to the rough part buffer)

Create a machine-sub-frame in the class library according to Fig. 4.43.



**Fig. 4.43** Sub-frame Machine

Enter in the machine a setup time of two hours. The init method creates an empty container on the station finish\_parts such as:

```
is
do
    .MUs.container.create(finish_parts);
end;
```

Create a frame from the machine (sub-frames) in the necessary number. Add for each machine group a buffer and connect these with the machines (Fig. 4.44).

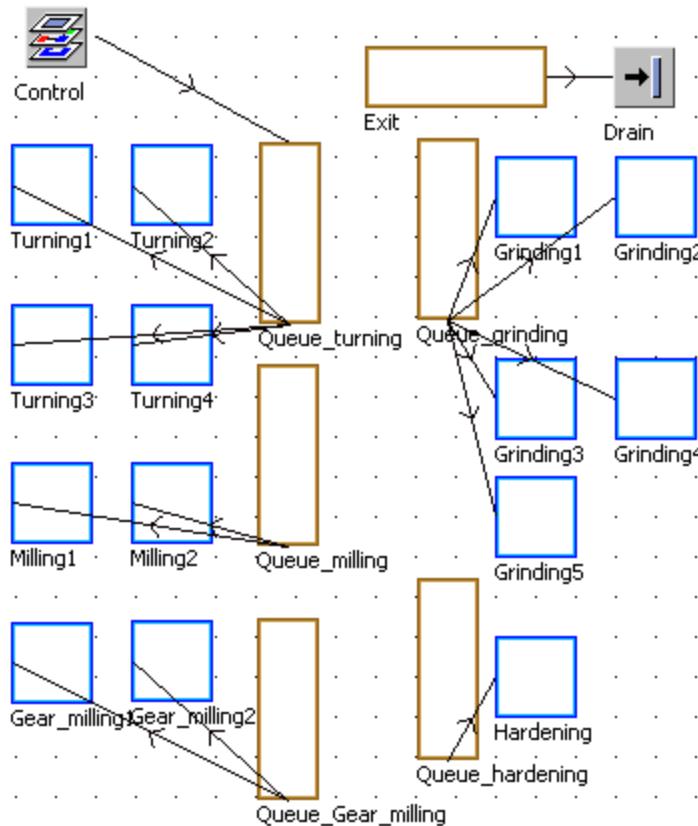


Fig. 4.44 Example Frame

#### 4.9.2 Elements of the Job Shop Simulation

Within job shop manufacturing is the rule that parts have different production processes. This makes it impossible to arrange the machines according to the flow principle and the parts must be transported from the workshop (e.g. turning) to the workshop analog of their production process. Connecting the machines using

connectors, therefore, is not an option for modeling. Instead, the next production step must be determined from the work plans and the transport will be initiated after completion of processing using SimTalk. To model a corresponding control, at least the following elements are required:

- work plan management
- order management
- resource management
- production control

#### 4.9.2.1 Work Plans

The work plan contains, for each part of the sequence of processing steps, the corresponding processing times and the required resources for the processing. The sources for this information are in many cases Excel or database tables in practice. For the programming of the control in Plant Simulation, the names of parts as well as of the resources must be unique. To store the work plans in Plant Simulation, a multi-dimensional table structure (table with sub-tables) is suitable. The work plan must contain at least the following information:

- Ordinal number (sort criterion for making a unique order)
- Name of the processing
- Processing time
- Necessary resource/machine

Insert in the example a sub-frame ("control"). Add in the frame "control" a "work\_plans" table. Format the table like in Fig. 4.45.

The screenshot shows a Plant Simulation interface with a main table and a modal dialog for a sub-table.

**Main Table:**

	string 0	table 1	integer 2	integer 3	real 4	string 5
string	Part	Workplan	bin_capacity	lot_size		
1	W1	x				
2	W2	x				
3						
4						
5						
6						
7						

**Modal Dialog (Sub-table for row 1):**

	string 1	string 2	time 3	string 4
string	OP	Operation	processing time	machine

Fig. 4.45 Work Plan

Add the following two work plans:

Part W1 (Fig. 4.46):

	string 1	string 2	time 3	string 4
string	OP	Operation	processing_time	machine
1	010	Turning	10:00.0000	turningmachine
2	020	Milling	5:00.0000	millingmachine
3	030	Hardening	2:30.0000	hardeningmachine
4	040	Grinding	5:00.0000	grindingmachine

**Fig. 4.46** Work Plan W1

Part W2 (Fig. 4.47):

	string 1	string 2	time 3	string 4
string	OP	Operation	processing time	Machine
1	010	Turning	5:00.0000	turningmachine
2	020	Gearmilling	10:00.0000	garmillingmachine
3	030	Hardening	2:30.0000	hardeningmachine
4	040	Grinding	15:00.0000	grindingmachine

**Fig. 4.47** Work Plan W2

For further modeling, we need two more part-specific data:

- Lot size
- Container (bin) capacity

The container capacity is important for the transport of the parts. Usually, parts in job shop manufacturing are not transported individually; they are transported in containers. Several containers form a lot, which is processed without interruption one after the other. Expand the table work\_plans in accordance with Fig. 4.48.

	string 0	table 1	integer 2	integer 3	r 4
string	part	workplan	bin capacity	lot size	
1	W1	x	50	400	
2	W2	x	50	400	

**Fig. 4.48** Table work\_plans

While the container capacity is relatively constant (depending on the size of the container), the lot size is often the subject of examination.

#### 4.9.2.2 Order Management

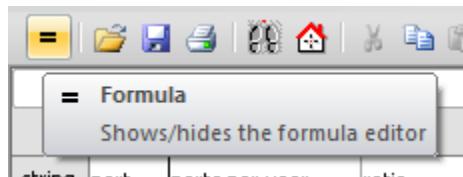
An order usually represents an instruction to the production to produce a certain amount of parts up to a certain date. Often, order data are supplied as quantity-time frames (number of products in a given time). Another possibility is providing a target program in the form of a distribution of the individual products to a total amount. From the available data, you must then create an "endless" program consisting of a series of individual lots (consisting of a certain number of containers) for the simulation. Furthermore, the order management must monitor the processing of the lots and provide statistical data after completion of the processing of orders.

Insert in the frame Control a "production\_program" table with the content from Fig. 4.49.

	string 0	integer 1	real 2	inte 3
string	part	parts per year	ratio	
1	W1	100000	1.00	
2	W2	200000	2.00	
-	-	-	-	-

**Fig. 4.49** TableFile production\_program

To calculate the ratio of the individual parts, you could divide each value of the column parts per year by the smallest value from the column parts per year. Plant Simulation offers the ability to create calculations within tables. To do this, place the cell pointer in the result field (e.g. first row, second column). Then click on the "=" symbol in the window of the table (Fig. 4.50).



**Fig. 4.50** TableFile—Formula Editor

The formula should appear as in Fig. 4.51.

	part	parts per year	ratio
	W1	100000	?[1,1]/? .min({1,1}..{1,*})
	W2	200000	0.00

**Fig. 4.51** Formula

You must now produce an "infinite" sequence of parts packaged in containers. It is helpful, in that case, to create an order list (TableFile order in sub-frame control). In the simplest form, the order list has two columns (Fig. 4.52).

	string 1	integer 2
string	part	number
1	W1	400

**Fig. 4.52** TableFile Orders

The row index of the order table represents the order number. Many companies work with alphanumeric order numbers (e.g. A000012012). In this case, you would need an additional column for the order number. In the case of equal lot sizes (for all parts), you can create a sequence of orders according to the following example (init method in the sub-frame control).

```

is
  i:integer;
  k:integer;
do
  -- delete entries in order table
  orders.delete;
  --set lot_size
  for i:=1 to work_plans.yDim loop
    work_plans[3,i]:=lotsize;
  next;
  -- create 10000 orders for each part
  for i:=1 to 10000 loop
    --for each line in production program
    --write part and ratio*lot_size in orders
    for k:=1 to production_program.yDim loop
      orders.writeRow(1,orders.yDim+1,
      production_program[0,k],
      production_program[2,k]*
      work_plans[3,production_program[0,k]]);
    next;
  next;
end;

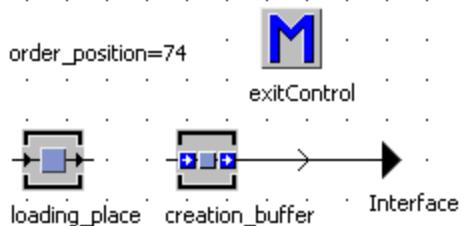
```

The Orders table now contains 20,000 individual production orders.

To produce the parts in the simulation, Plant Simulation provides the source. In job shop manufacturing, some circumstances require their own special source:

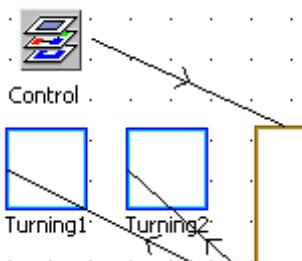
- The generation timings are variable, some processes may "pull" the parts from the source; the source must produce parts when the parts are required
- The generation parameters must remain variable and may vary from experiment to experiment (number per container, sequence of parts)

With a little effort, you can develop your own object for this purpose. To simulate a "pulling" by the production, the object needs its own "creation buffer." The creation buffer is connected to the buffer of the first production machine (often marking). When the creation buffer is empty, container and parts are generated according to the next position in the production sequence. Since the generation of parts needs to access all data from the "Control" network, we will build the necessary structure in this frame. Add the objects from Fig. 4.53 into the frame control.



**Fig. 4.53** Extension Sub-frame Control

The variable `order_position` has the data type integer and an initial value of zero. The method `exitControl` is the exit control (rear) of the `creation_buffer`. The method should always be called when the buffer is empty and should produce the next lot according to the order list. The `loading_place` first holds the empty container. This is filled with parts (up to the prescribed number). It is then moved to the `creation_buffer`. Through the `Interface`, containers are transported to the first buffer in the processing sequence (by connecting the control network with the first buffer—in the present example, the buffer of turning).



**Fig. 4.54** Connection Control Sub-frame

The `exitControl` (sub-frame control) could look like this:

```
is
  part:string;
  number:integer;
  bin_capa:integer;
  counter:integer;
```

```

i:integer;
bin:object;
t:object;
do
  if creation_buffer.empty then
    --next order position
    order_position:=order_position+1;
    if order_position > orders.yDim then
      order_position:=1;
    end;
    --read data from order table
    part:=orders[1,order_position];
    number:=orders[2,order_position];
    bin_capa:=work_plans[2,part];
    -- create bins and parts until
    --number = counter
    counter:=0;
    while counter < number loop
      --create container on loading_place
      bin:=.MUs.container.create/loading_place);
      bin.xDim:=bin_capa;
      for i:=1 to bin_capa loop
        t:=.MUs.part.create(bin);
        t.name:=part;
        counter:=counter+1;
        if counter=number then
          exitloop;
        end;
      next;
      --move
      bin.move(creation_buffer);
    end;
  end;
end;

```

The parts are named W1 and W2.

#### 4.9.2.3 Resource Management

In order to work with the work plans of the parts, references on the real objects in Plant Simulation must be stored. For production control, assigning the machine names from the work plans to the machine networks and their upstream buffers is needed. An appropriate resource table might have the following form (TableFile resources in sub-frame control, Fig. 4.55)

	object 1	string 2	object 3
string	Machine_object	description	buffer
1	root.Turning1	turninigmachine	root.Queue_turning
2	root.Turning2	turninigmachine	root.Queue_turning
3	root.Turning3	turninigmachine	root.Queue_turning

Fig. 4.55 TableFile Resources

Enter the information for all machines into the resource table.

#### 4.9.2.4 Production Control

In the simplest form, the production control system must include two elements:

- Set the processing times on the machines, load machines and unload finished parts into the finished-part containers
- After processing: Find the next machine in the work plan, transfer the container into the associated buffer

The biggest part of the production control can be programmed in the machine object. The following two additional controls are required:

- ExitControl (Front) green\_parts (for loading the machine)
- ExitControl (Front) machine (for unloading the machine into the finished-part container, loading the machine and transferring to the next station)

To control the processing of the parts, the parts need a custom attribute OP (integer) and the initial value is 1. The OP attribute stores the position (line number) within the work plan for each part. After each operation, this attribute is incremented by one. The ExitControl of the green\_parts place determines the processing time for the part type in the container (column 3, line OP in the work plan), sets the processing time of the machine and loads the first part onto the machine:

```

is
  part:string;
  proc_time:time;
  op:integer;
do
  part:=@.cont.name;
  op:=@.cont.op;
  --set capacity of finish-parts container
  finish_parts.cont.xDim:=@.xDim;
  --read procTime from Workplan
  proc_time:=root.control.work_plans[1,part][3,op];
  --set procTime of machine
  machine.procTime:=proc_time;

```

```
--move the first part to the machine
@.cont.move(machine);
end;
```

After the machining on the machine, the OP attribute of the part is increased by one, and the part is moved into the finished-part container. The next part is loaded from the green-part container to the machine. If the part was the last out of the container (container on the green-parts place is empty), the next buffer is determined (machine name of the next OP → browse for the corresponding buffer in the resource table); if no machine is found (last OP), then the part will be placed on exit. If the last part has been placed in the finished part container, the finished part container is relocated to the next buffer. The green-part container will be relocated to the finished-part place. The next incoming container with green parts starts the subsequent processing. The ExitControl of the machine inside the machine network should look like this:

```
is
  target:object;
  next_machine:string;
do
  --increase OP by one
  @.op:=@.op+1;
  --if the last part from the bin --> read target
  if green_parts.cont.empty then
    next_machine:=
      root.control.work_plans[1,@.name][4,@.op];
    --lookfor the machine in the second column
    if next_machine /= "" then
      root.control.resources.setCursor(2,1);
      root.control.resources.find({2,1}..{2,*},
        next_machine);
      target:=root.control.resources[3,
        root.control.resources.cursorY];
    else
      target:=root.exit;
    end;
    @.move(finish_parts.cont);
    --to the next machine
    finish_parts.cont.move(target);
    waituntil finish_parts.empty prio 1;
    --move empty bin from green_parts to finish_parts
    green_parts.cont.move(finish_parts);
  else
    @.move(finish_parts.cont);
    --get next part for machine
    green_parts.cont.cont.move(machine);
  end;
end;
```

## 4.10 Pull Control

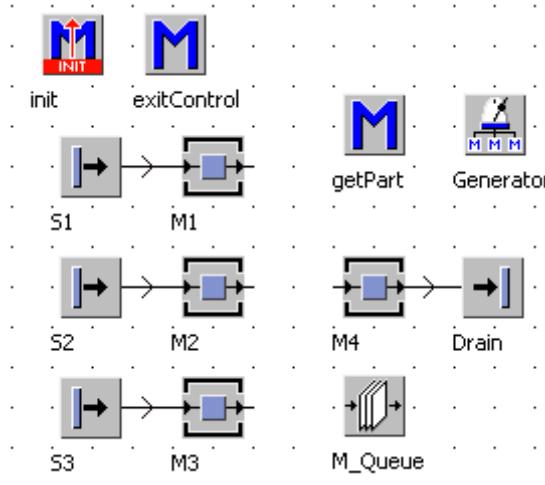
In classic pull control, the transport of the MUs is not based on a fixed schedule nor does it run automatically along connectors. Instead, it is triggered by the succeeding station.

### 4.10.1 Simple Pull Control

You can relatively easily create pull controls using a generator and a method. The generator periodically tests the precondition for moving the MUs and releases it when all conditions are met.

#### Example: Pull Control

In the following example, a machine processes parts from three upstream workstations according to the FIFO principle. Create a frame as in Fig. 4.56.



**Fig. 4.56** Example Frame

Ensure the following settings: M1, M2 and M3 have a processing time of three minutes each, and an availability of 95 per cent and 10 minutes of MTTR. Assign the method exitControl as the exit control (front) for M1, M2 and M3. M4 has a processing time of one minute, an availability of 95 per cent and 10 minutes of MTTR. The QueueFile M\_Queue has the data type object. Every 10 seconds, the generator calls the method getPart. The init method deletes the contents of M\_Queue:

```
is
do
  m_Queue.delete;
end;
```

The FIFO principle should be implemented in the following way: If an MU is finished at station M1, M2, M3, it triggers the exit control. The exit control enters the MUs

sequentially in the QueueFile M\_Queue. The method getPart then moves the respective first MU in the queue to M4. Therefore, the exitControl has the following content:

```
is
do
    M_Queue.push(?);
end;
```

Method getPart:

```
is
    m:object;
do
    if M_Queue.dim> 0 and M4.empty and
        M4.operational then
        m:=M_Queue.pop;
        m.cont.move(M4);
    end;
end;
```

## 4.10.2 Kanban

Kanban is a method of a self-controlled production that follows the pick-up principle. The flow of material here is directed forward (from producer to consumer), while the flow of information is directed backward (from consumer to producer). Permanent interventions of a central control are unnecessary in the Kanban system. The Kanban system is, in the strict sense, an information system to control production processes harmoniously and efficiently. Kanban control is often called a supermarket principle. In a supermarket, goods are available for purchase for consumers. The consumer removes the required items from the shelf, and the staff of the supermarket fills the shelves as needed again. Usually, the sales staff removes the goods from an intermediate storage at the supermarket. This increases inventory, which gives the system security, but it also makes processes more expensive. In some supermarkets, there is no intermediate storage; suppliers directly fill the shelves. However, this process control depends on geographic distances, delivery times and customer demands. In a Kanban control, this principle is transferred to a production process—e.g.:

- The assembly of a company manufactures products and takes all necessary components from a shelf (warehouse).
- The upstream departments or suppliers on their own fill the shelves (warehouse) again.

### 4.10.2.1 Functioning of the Kanban System

In the Kanban system, plan-oriented job control is replaced by a consumption-controlled job control by small, interrelated, self-regulating loops between each successive step in the workflow.

The main elements of this system are:

- Creation of linked, self-regulating control loops between producer and consumer areas
- Implementation of the pick-up principle for each subsequent consumption level
- Flexible personnel and resource use
- Transfer of short-term control to the executive staff
- Using a special information carrier, the Kanban card is specially developed for this purpose and the associated standard Kanban containers

#### 4.10.2.2 Control Loops

There are two main control loops in the Kanban system:

- a) Consumer—warehouse (transport Kanban)
- b) Consumer/warehouse—production (production Kanban)

In the simulation, we can model Kanban cards virtually (e-Kanban; the information is electronically transmitted between supplier and consumer) or set up your own "material flow" for the Kanban cards. The effectiveness of a Kanban system is, among other things, measured by the number of Kanban cards used. If the number of Kanban cards is one of the KPI to be detected, then the Kanban cards themselves must be simulated as MU.

#### 4.10.2.3 Modeling of a Single-Stage E-Kanban System

Within an e-Kanban system, the physical (Kanban) card is replaced by a purely technical information message. In principle, the consumption point directly triggers an order for the supplier (where the supplier can also be a warehouse)—e.g. after removal of the first part from a container. This order must contain the following information so that delivery can be made without problems:

- Part number, possibly description
- Kind of transport container to be used
- Number of parts per container
- Supplier
- Consuming point

The supplier must immediately after receipt of the e-Kanban send the number of parts in the specified container to the consuming point. Since the material flow is not coupled with the Kanban flow, there is usually an independent empty-container flow (without order reference) and additional empty-container storage. The following example illustrates the modeling of a simple one-stage e-Kanban system. The transport processes should be taken into account by general transport times.

#### Example: E-Kanban

On two assembly stations, five different parts are produced in lot size one. There are three final products with the following BOMs:

W1		W2		W3	
Part	Number	Part	Number	Part	Number
T1	10	T1	5	T1	20
		T2	3	T2	5
		T4	12	T3	5
				T4	25

The parts are supplied from an internal warehouse and are delivered in standardized containers of different capacities:

Part	Container capacity
T1	1000
T2	100
T3	100
T4	500
T5	200

The following cycle times and shares in the total amount apply here:

W1: 1:30 (60 per cent)

W2: 5:00 (30 per cent)

W3: 10:00 (10 per cent)

For each part, type two container places (Buffer) are required. Deliveries will be made in principle after taking the first part out of a full container (during removal from a container). Create a frame according to Fig. 4.57.

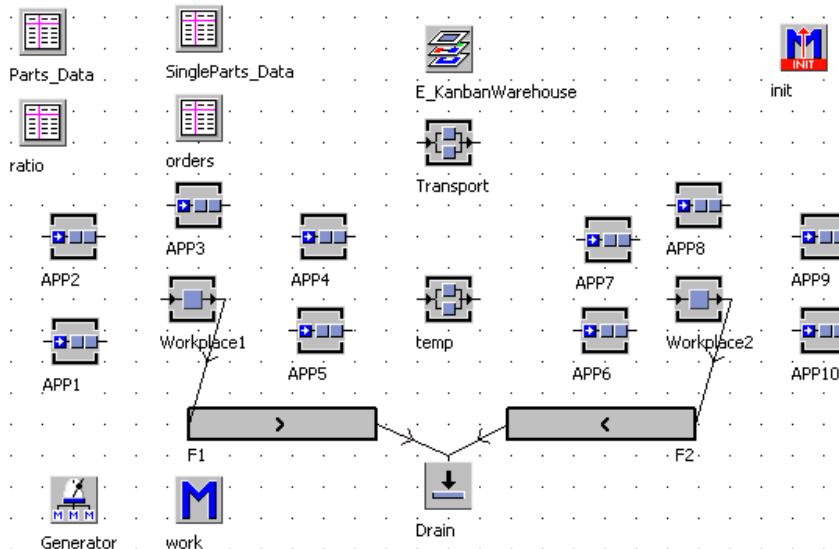


Fig. 4.57 Example Frame

First, some information must be stored in a suitable form in the simulation.

*Bill of Material (BOM) and Cycle Times*

BOM and cycle time can be stored most easily in a table with sub-tables (Table Parts\_Data). For every part, one needs to store the items, the quantities of the items and the assembly time. A structure according to Fig. 4.58 is suitable. Insert the data into the table.

x						
	string 0	table 1	time 2	string 1	integer 2	string 3
string	assembly	BOM	assembly time	string	Item	Number
1	W1	x	1:30:0000	1	T1	5
2	W2	x	5:00:0000	2	T2	3
3	W3	x	10:00:0000	3	T4	12
				4		

Fig. 4.58 TableFile Parts\_Data

The TableFile SingleParts\_Data contains the container capacities of the individual parts. Enter the relevant information analogous to Fig. 4.59.

	string 0	integer 1
string	part	container capacity
1	T1	1000
2	T2	100
3	T3	100
4	T4	500
5	T5	200

Fig. 4.59 TableFile SinglePart\_Data

*Order List*

The order list (W1, W2, W3 in batch size one; random order from distribution key) can be created using the method z\_dEmp and a distribution table. Format and fill in the distribution table "ratio" as shown in Fig. 4.60.

	string 0	integer 1	real 2
string	Part	Position	Ratio
1	W1	1	60.00
2	W2	2	30.00
3	W3	3	10.00

Fig. 4.60 TableFile Ratio

The distribution table of the method `z_dEmp` can contain in the first column only the data types that are integer, real, or time; hence, we need to take a little detour via an index for determining the parts.

The order list itself needs a variable to store the current position within the list, and a method for initially filling the order list. Insert in the `TableFile orders` user-defined attributes according to Fig. 4.61.

Name	Value	Type
<code>fill</code>		method
<code>getNextPosition</code>		method
<code>numOrders</code>	25000	integer
<code>order_position</code>	1	integer

Fig. 4.61 User-defined Attributes in `TableFile Orders`

Set `numOrders` to 25,000. To fill the order list, we call the `z_dEmp` function 25,000 times and write the relevant parts successively into the list. In this way, we get a random sequence with the required percentages. Method `orders.fill`:

```
is
  i:integer;
  part:string;
do
  for i:=1 to self.~.numOrders loop
    --column 0 contains the part
    --row is calculated with dEmp
    part:=self.~.ratio[0,z_dEmp(1,self.~.ratio)];
    self.~.writeRow(1,i,part);
  next;
end;
```

The method `orders.getNextPosition` calculates and returns the next order position.

```
:integer
is
do
  self.~.order_position:=self.~.order_position+1;
  if self.~.order_position > self.~.yDim then
    self.~.order_position:=1;
  end;
  return self.~.order_position;
end;
```

#### *Assembly Place, Part Consumption*

To simplify the programming of the logic, further preparatory steps are necessary. To create different MUs for the finished products, it is advantageous to store a reference to the MUs to be generated (class library) in the `parts_data TableFile`.

Create three different entities in the class library (W1, W2, W3) and extend the TableFile parts\_data according to Fig. 4.62.

	string 0	table 1	time 2	object 3
string	assembly	BOM	assembly time	MU
1	W1	x	1:30.0000	.ApplicationObjects.EKanban.MUs.W1
2	W2	x	5:00.0000	.ApplicationObjects.EKanban.MUs.W2
3	W3	x	10:00.0000	.ApplicationObjects.EKanban.MUs.W3

Fig. 4.62 TableFile parts\_data

The MUs can now be produced easily.

The workstations require the combination of an item name and a corresponding container buffer. To do this, insert in each workplace a user-defined attribute (BufferTable, data type table) according to Fig. 4.63 (example data of workplace1).

	string 0	object 1
string	part	buffer
1	T1	APP1
2	T2	APP2
3	T3	APP3
4	T4	APP4
5	T5	APP5

Fig. 4.63 Buffer Table

Assign the relevant buffers per workplace.

Add in each workstation a user-defined attribute "work" (data type method). The workplace in the simulation must display the following behavior: If the workplace is empty, orders must still be processed and all part buffers of the workplace must be occupied; then the workplace will read the next job from the Orders table and determine the parts to be consumed from the bill of material. A new MU (assembly) is created on the workstation. The parts are removed (destroyed) from the relevant containers. When the container is empty, it is transferred to the block transport. The finished part is relocated on the conveyor lines F1 or F2. The method "work" of workplace1 should appear as follows:

```
is
  finish_part:string;
  finish_part_mu:object;
  mu:object;
  buffer:object;
```

```

number:integer;
single_part:string;
i:integer;
k:integer;
remain:integer;
order_position:integer;
do
  --return, if the workplace is occupied or
  --one of the buffers is not occupied
  if self.~.occupied then
    return;
  end;
  for i:=1 to self.~.bufferTable.yDim loop
    if self.~.bufferTable[1,i].empty then
      return;
    end;
  next;
  --next order, read part, increase order_position
  order_position:=root.orders.getNextPosition;
  finish_part:=root.orders[1,order_position];
  --create finish_part on temp
  mu:=root.parts_data[3,finish_part];
  finish_part_mu:=mu.create(temp);
  finish_part_mu.order_position:=order_position;
  --set processing time
  self.~.procTime:=root.parts_data[2,finish_part];
  --move part from temp to self.~
  finish_part_mu.move(self.~);
  --delete all single parts from the BOM
  for i:=1 to
    root.parts_data[1,finish_part].yDim loop
      single_part:=root.parts_data[1,finish_part][1,i];
      number:=root.parts_data[1,finish_part][2,i];
      buffer:=self.~.bufferTable[1,single_part];
      --enough parts for assembly?
      if buffer.cont.numMu >= number then
        if buffer.cont.full then
          --send E-Kanban, before we take
          -- the first part from the container
        end;
        --delete number of parts
        for k:=1 to number loop
          buffer.cont.cont.delete;
        next;
        --move empty container to transport
        if buffer.cont.empty then
          buffer.cont.move(transport);
        end;
      end;
    end;
  end;
end;

```

```

    end;
else
    --remaining number of parts is too
    --low for assembly
    remain:=number-buffer.cont.numMu;
    --first take te remaining parts
    buffer.cont.deleteMovables;
    --move empty container to transport
    buffer.cont.move(transport);
    --some parts from the next container
    --send E-Kanban
    for k:=1 to remain loop
        buffer.cont.cont.delete;
    next;
    end;
next;
end;

```

Note: The parts are first generated in an auxiliary place (temp). If you want to set processing times dynamically, you must do so before the part is relocated to the station. Therefore, the processing time is set first and then the finished part is moved to the workstation. When you create the part directly on the workplace, the part is immediately ready for exit (without processing time).

The method "work" of the frame is called by the generator every 10 seconds. To do this, connect the work method in the frame with the generator. The method work in the frame should look like this:

```

is
do
    workplace1.work;
    workplace2.work;
end;

```

Each workstation must be initialized. For this, in each of the attached buffers, a filled container must be created. An internal method attribute (method init) of the workplace could look like this (for all workstations):

```

is
    buffer:object;
    single_part:string;
    bin:object;
    part:object;
    i:integer;
    number:integer;
do
    for i:=1 to self.~.bufferTable.yDim loop

```

```

buffer:=self.~.bufferTable[1,i];
single_part:=self.~.bufferTable[0,i];
number:=SingleParts_Data[1,single_part];
--new bin
bin:=.ApplicationObjects.EKanban.MUs.bin.create(
    buffer);
--set capa
bin.xDim:=number;
--fill
while bin.full=false loop
part:=.ApplicationObjects.EKanban.MUs.T.create(
    bin);
part.name:=single_part;
end;
next;
end;

```

**Note:** In order to facilitate the customizing of the capacity, set the property yDim of the container to one. Now, you can set the capacity of the container alone by setting the xDim property.

### *Transport*

Especially if transport is not the focus of the simulation, you can model it by considering an average transport time. With the help of a sufficiently large parallel station (e.g. 20 places) on which the average transport time (e.g. five minutes) is set, you can easily simulate transport. The transport block has an exit control (custom method-attribute exitControl, front). In the simulation, empty containers are always moved into the empty container storage of the E\_KanbanWarehouse network and full containers to the specified destination. To address the full container, you can use the attribute destination of the container. The exitControl of transport must appear as follows:

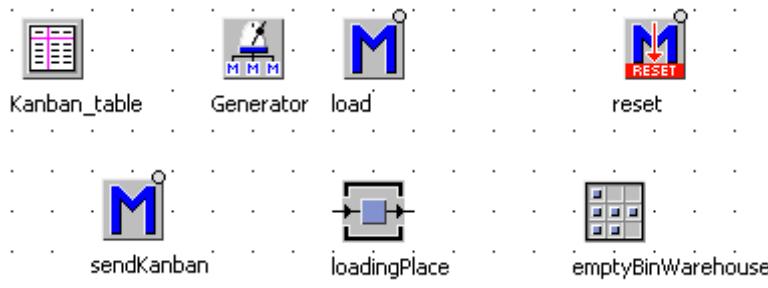
```

is
do
  if @.empty then
    @.move(E_KanbanWarehouse.emptyBinWarehouse);
  else
    @.move(@.destination);
  end;
end;

```

### *The Sub-frame E\_KanbanWarehouse*

All Kanban functions are encapsulated in the frame E\_KanbanWarehouse. Create the content of the frame as shown in Fig. 4.64.



**Fig. 4.64 E\_KanbanWarehouse**

In the first step, we do without the consideration of the warehouse. The table Kanban\_table stores all e-Kanban requests. The table must contain at least two columns: part (string) and destination (object). The generator calls each 5 seconds the method load. The method sendKanban stores the e-Kanban information in the table Kanban\_table:

```
(part:string;target:object)
is
do
  kanban_table.writeRow(1, kanban_table.yDim+1,
    part,target);
end;
```

The method load monitors the Kanban\_table and fills in the loading place a container with the required type and number of items. The method then addresses the container and relocates the container to the transport station:

```
is
  bin:object;
  number:integer;
  part:string;
  target:object;
  part_mu:object;
do
  --wait for E-Kanban
  if kanban_table.yDim = 0 or
    loadingPlace.occupied then
    return;
  end;
  --read data and delete row
  part:=kanban_table[1,1];
  target:=kanban_table[2,1];
  kanban_table.cutRow(1);
  --read number from parts_data
  number:=self.~.~.SingleParts_data[1,part];
```

```

--if warehouse occupied then move container
-- from warehouse
--else create a new container
if emptyBinWarehouse.occupied then
  bin:=emptyBinWarehouse.cont;
  bin.move(loadingPlace);
else
  bin:=.ApplicationObjects.EKanban.MUs.bin.create(
    loadingPlace);
end;
--set capacity
bin.xDim:=number;
--fill
while bin.full=false loop
  part_mu:=
    .ApplicationObjects.EKanban.MUs.T.create(bin);
  part_mu.name:=part;
end;
--adress bin
bin.destination:=target;
--move to transport
bin.move(self.~.~.transport);
end;

```

The reset method deletes all old entries in the Kanban\_table:

```

is
do
  kanban_table.delete;
end;

```

### *Kanban Control*

The Kanban control consists of a combination of different methods.

1. After removal of the first part of a container, an e-Kanban is sent to the "E-Kanban-Mailbox" (methods workplace.work+ E\_KanbanWarehouse.sendKanban)
2. The method E\_KanbanWarehouse.Load monitors the "E-Kanban-Mailbox" (Kanban\_table)
3. If an e-Kanban has arrived (FIFO), the method E\_KanbanWarehouse.Load loads the container according to the specifications in the "E-Kanban mailbox".
4. The filled container is addressed and handed over to the transport system (E\_KanbanWarehouse.Load).
5. The exitControl of the transport station "transports" the filled containers to the correct buffer.

To complete the control, the sending of the e-Kanban must be integrated into the work methods of the workstations in two places:

- After removing the first part from a container:

```

if buffer.cont.numMu >= number then
  if buffer.cont.full then
    E_KanbanWarehouse.sendKanban(single_part, buffer);
  end;
-- and so on
- after changing the container

--some parts from the next bin
--send E-Kanban
E_KanbanWarehouse.sendKanban(single_part,buffer);

```

#### 4.10.2.4 Bin Kanban System

Within a bin Kanban system, the bin carries the Kanban information. There is no flow of information separated from the bin. The Kanban control would appear as follows:

1. A bin is emptied at the consumption point.
2. The empty bin is transported to the filling up point (production, warehouse).
3. The container is refilled with reference to the bin Kanban information.
4. The filled bin is transported to its point of consumption.

Compared to the e-Kanban system, the process from triggering the request until the refilling takes longer in the bin Kanban-System because the process now includes the transport of the empty bin.

#### Example: Bin Kanban System

The following example is based on the example of e-Kanban (you can download it from <http://www.bangsow.de>).

##### *Initializing*

The initialization of a bin Kanban system must always occur with several bins for each buffer area, since the filling of the bin is triggered only when the bin is completely empty. Initialization with only one bin would result in an interruption of the production flow. To do this, modify the init methods of the workstations, so that in each buffer two bins are produced:

```

is
  buffer:object;
  single_part:string;
  bin:object;
  part:object;
  i:integer;
  number:integer;
do
  for i:=1 to self.~.bufferTable.yDim loop
    buffer:=self.~.bufferTable[1,i];

```

```

single_part:=self.~.bufferTable[0,i];
number:=SingleParts_Data[1,single_part];
--new bin
bin:=.ApplicationObjects.EKanban.MUs.bin.create(
    buffer);
--set capa
bin.xDim:=number;
--fill
while bin.full=false loop
    part:-
        .ApplicationObjects.EKanban.MUs.T.create(
            bin);
    part.name:=single_part;
end;
--second bin
bin:=
    .ApplicationObjects.EKanban.MUs.bin.create(
        buffer);
--set capa
bin.xDim:=number;
--fill
while bin.full=false loop
    part:-
        .ApplicationObjects.EKanban.MUs.T.create(bin);
    part.name:=single_part;
end;
next;
end;

```

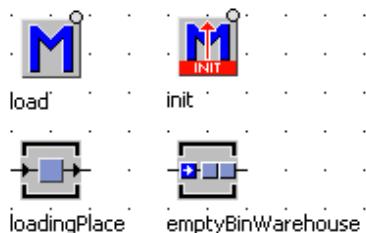
#### *Correction of the Method Work of the Workplace*

In contrast to e-Kanban, a Kanban is not sent; remove the lines:

```
--send E-Kanban
E_KanbanWarehouse.sendKanban(single_part,buffer);
```

#### *The Sub-frame Bin\_KanbanWarehouse*

Create a sub-frame with the elements from Fig. 4.65.



**Fig. 4.65** Elements of Bin\_KanbanWarehouse

Replace the sub-frame E\_KanbanWarehouse with the sub-frame Bin\_KanbanWarehouse (Fig. 4.66).



**Fig. 4.66** Example Frame

#### *Kanban Control*

The container must be able to store the Kanban information. At least, these are the following details:

- Part (string)
- Number (integer)
- Consumption point (object)

Add to this the filling point when there are various warehouse locations or places of production. Create user-defined attributes in the container (class library) as in Fig. 4.67.

Name	Value	Type
consumption_point	(?)	object
number	0	integer
part		string

**Fig. 4.67** Bin Attributes

The attributes must be set once at the beginning of the simulation. Then, "shuttle" the containers between supplier and consumer. We need to integrate the set of attributes in the init methods of the workplaces (for the two bins):

```
bin:=
.ApplicationObjects.EKanban.MUs.bin.create(buffer);
--set capa
bin.xDim:=number;
--set attributes
bin.number:=number;
bin.consumption_point:=buffer;
bin.part:=single_part;
-- and so on
```

We must correct the exitControl of the transport station (destination of empty bins, now the sub-frame Bin\_KanbanWarehouse)

```

is
do
  if @.empty then
    @.move(Bin_KanbanWarehouse.emptyBinWarehouse);
  else
    @.move(@.destination);
  end;
end;

```

The method load in the sub-frame Bin\_KanbanWarehouse must model the following behavior: The method waits until a bin arrives in the emptyBinWarehouse (the buffer is occupied). Next, the method moves the bin to the loadingPlace, reads all Kanban information and generates the correct number of parts in the bin. The method addresses the bin and transfers it to the block transport. Finally, the method calls itself, in order to be able to respond to the next bin. The method load should, therefore, have the following programming:

```

is
  bin:object;
  number:integer;
  part:string;
  destination:object;
  part_mu:object;
do
  --wait for the next bin
  waituntil emptyBinWarehouse.occupied and
    loadingPlace.empty prio 1;
  --move bin
  bin:=emptyBinWarehouse.cont;
  bin.move/loadingPlace);
  --read info
  part:=bin.part;
  destination:=bin.consumption_point;
  number:=bin.number;
  --adopt bin capacity
  bin.xDim:=number;
  --fill the bin
  while bin.full=false loop
    part_mu:-
      .ApplicationObjects.EKanban.MUs.T.create(bin);
    part_mu.name:=part;
  end;
  --address bin
  bin.destination:=destination;
  --move to transport

```

```
bin.move(self.~.~.transport);  
--call  
ref(load).methCall(0.1);  
end;
```

The init method of the sub-frame Bin\_KanbanWarehouse calls at the beginning of the simulation the method load:

```
is  
do  
  load;  
end;
```

The example can now be expanded as required.

#### 4.10.2.5 Card Kanban System

The classic Kanban system separates empty bin flow and information flow. The cards are transported separately. The procedure is analogous to the e-Kanban system with the difference that the cards are physically located in the Kanban mailbox and must be taken from there. The challenge for the simulation is the modeling of the physical movement of Kanban cards as their own "information flow." Usually, however, you need not model the Kanban card flow; instead, you can model the system as an e-Kanban system.

#### 4.10.3 The Plant Simulation Kanban Library

Plant Simulation has its own blocks for modeling a Kanban control. You need to include these elements in the class library: File—Manage Class Library Tab Libraries—mark Kanban (Fig. 4.68).

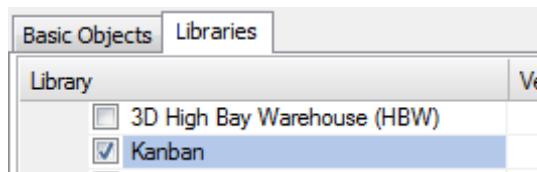


Fig. 4.68 Kanban Library

#### Example: Plant Simulation Kanban Library

You should model the following scenario. Three parts are produced in a random sequence on a machine. The blanks are stored in buffer storage. If the buffer stock falls below a certain stock, it should be filled up by a source. In the simulation, the machine requests the blanks from the buffer storage. The buffer storage, in turn, requests the blanks from the source. Create a frame according to Fig. 4.69 using the Plant Simulation Kanban objects.

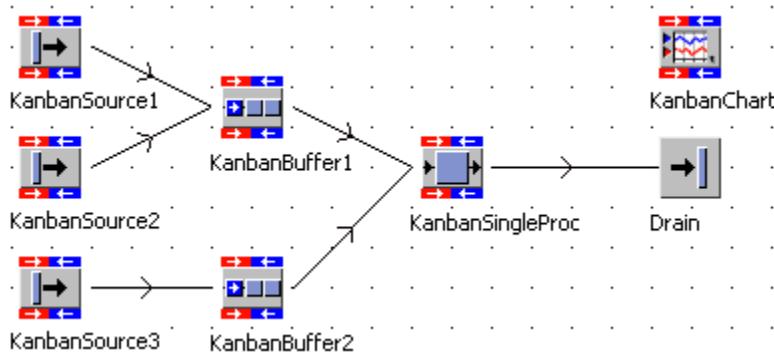


Fig. 4.69 Example Frame

Insert three entities in the class library (Part1, Part2 and Part3) and color them differently. The KanbanSingleProc manages the production plan and requests parts from the elements KanbanBuffer or KanbanSource. In the example, Part1, Part2 and Part3 will be manufactured in a distribution of 50, 30 and 20 per cent. The cycle time is two minutes.

Since the entire control of the production flow starts from the KanbanSingleProc, we will begin here with the configurations. In the tab Kanban information, specify the production program of the station. There are three options:

- Sequence (tab Advanced Sequence): You can define the order of the parts to be produced in a list. Processing is terminated at the end of the list.
- Cyclically repeated sequence: The input sequence is repeated endlessly.
- Random sequence: The parts are produced randomly based on a percentage distribution.

We choose the variant c. You must set the tab Kanban information, the parts to be produced, the distribution and the supplier of the parts. Insert values like in Fig. 4.70.

Entity Type	Portion	Supplier
Part1	50	KanbanBuffer1
Part2	30	KanbanBuffer1
Part3	20	KanbanBuffer2

Fig. 4.70 Kanban Information

You gain access to the underlying single station via the Open Workplace button on the Advanced tab. Set a processing time of two minutes in the single station. In the next step, we need to configure the KanbanBuffer objects so that they can deliver the requested parts. For each part, you must create the following settings:

- Entity type
- Minimal stock

- Maximal stock
- Initial stock
- Supplier

Clicking on the New Part button in the tab Kanban Information opens an input menu (Fig. 4.71).

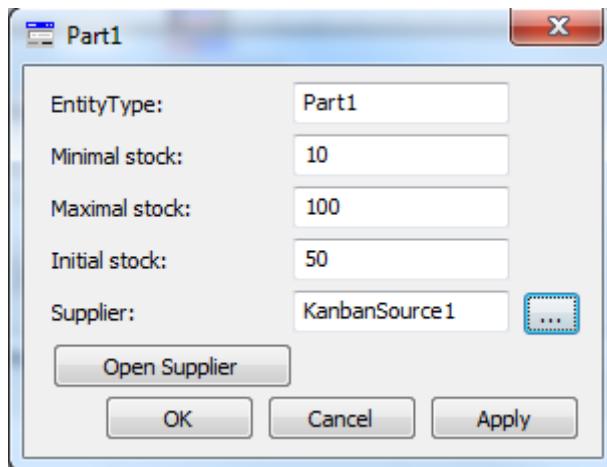


Fig. 4.71 Kanban Information

Enter the following values:

Entity type	Minimal stock	Maximal stock	Initial stock	Supplier
Part1	10	100	50	KanbanSource1
Part2	50	500	200	KanbanSource2
Part3	20	200	200	KanbanSource3

Note that Part3 is supplied from KanbanBuffer2. When reaching the minimal stock, the KanbanBuffer orders from the supplier the number of parts needed to fill up the buffer to the maximum stock. Now, in the KanbanSource objects, set which real MUs are to be produced on demand (Fig. 4.72).

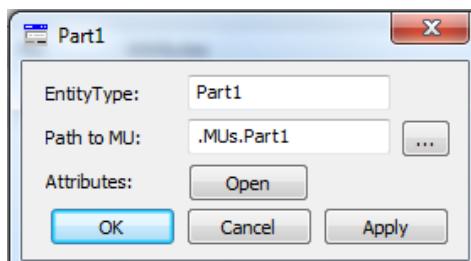


Fig. 4.72 Setting KanbanSource

Assign all MUs to be generated to the KanbanSource objects. The model now works with Kanban control. You can use the object KanbanChart to display the stock of the buffer objects graphically. Insert a KanbanChart block into the frame for each buffer. Then drag the KanbanBuffer objects onto the KanbanChart objects. For a better representation, we need to change some settings in the underlying chart block. Select the KanbanChart object Tools—Open chart object. Enter in the tab Axes, the number of values as 10,000 and an x-range of two days. The KanbanChart object now plots the stocks of parts in the buffer (Fig. 4.73).

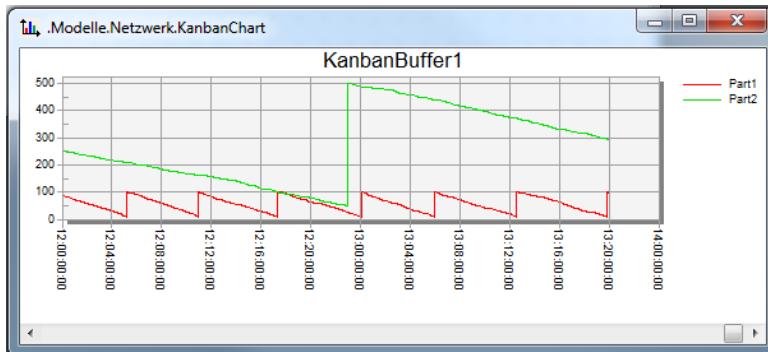


Fig. 4.73 KanbanChart

## 4.11 Line Production

Line productions, can be easily modeled in Plant Simulation. The interconnecting of a number of stations and conveyor technology elements using connectors already ensures the needed functionality. Even with these simple models, you can conduct practical investigations.

### 4.11.1 CONWIP Control

The object of the CONWIP (constant work in progress) control is the reduction of buffer stocks and, thus, of the throughput time of orders, by controlling the order release based on the number of order units in the system. In the simplest form, the order will get a CONWIP card that accompanies the order through the entire run of the job. Within the production system, a certain amount of work is fixed (e.g. maximum number of orders in the system = maximum number of CONWIP cards). A new order can only start in a filled system when a CONWIP card from another order becomes available (an order is completed). Another approach to control the WIP may be the necessary capacity to process an order (e.g. process time, workload of the bottleneck capacity). In this variant, an order cannot start until the necessary capacity in the system has become free.

### Example: CONWIP Control

Create a frame according to Fig. 4.74.

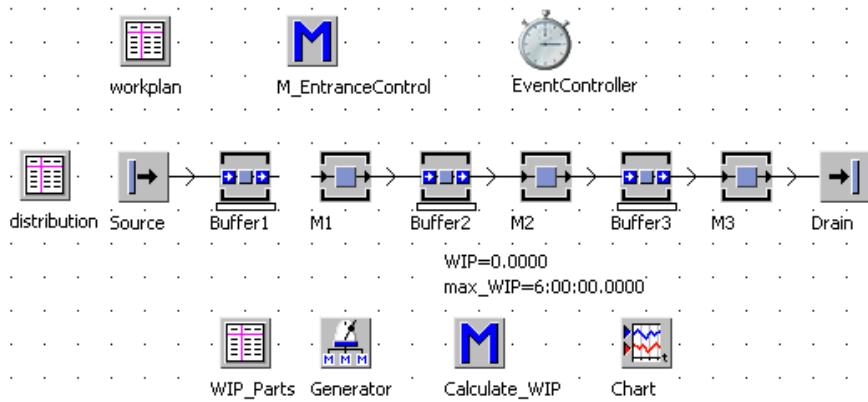


Fig. 4.74 Example Frame

Insert two parts in the class library (part1 and part2). Add in the parts a user-defined attribute op (integer, initial value = zero). The source generates Part1 and Part2 randomly with a distribution of 30 per cent and 70 per cent at intervals of 10 minutes. Buffer1 has a capacity of 1,000, Buffer2 and Buffer3 of 100 each. All buffers have no processing time. In the table workplan, the processing times of the parts are stored. Format as in Fig. 4.75.

	string 0	table 1		object 1	time 2
string	Part	Workplan		string	Machine
1	Part1	x		1	10:00.0000
2	Part2	x		2	15:00.0000
3				3	5:00.0000

Fig. 4.75 TableFile Workplan

Fig. 4.75 shows the data for Part1; Part2 has the following work plan:

M1: 2:00

M2: 5:00

M3: 5:00

Activate the failures in the SingleProcs: 90 per cent availability, 1:30:00 MTTR based on the simulation time. The method M\_EntranceControl is the entrance control (before actions) of the SingleProcs M1, M2 and M3. It reads and sets the processing times for the stations depending on part and operation, and increases the attribute op of the parts by one:

```

is
do
  --set processing time
  ?.procTime:=workplan[1,@.name][2,@.op+1];
  --increase op of the part
  @.op:=@.op+1;
end;

```

The table WIP\_Parts includes all parts (as a reference to the class) that should be included in the calculation of WIP (Fig. 4.76).

	object 1
string	Teile
1	.MUs.Part1
2	.MUs.Part2

Fig. 4.76 WIP Parts

The method calculate\_WIP calculates at regular intervals the amount of WIP (based on the sum of the processing times of the parts). If the necessary WIP for the next part is available in the system, the next part of Buffer1 is transferred to M1. Configure the generator so that it calls calculate\_WIP every five seconds. The Calculate\_WIP method might look like this:

```

is
  i,k:integer;
  part:object;
do
  WIP:=0;
  --calculate wip form all parts of wip_parts
  for i:=1 to WIP_Parts.yDim loop
    part:=WIP_Parts[1,i];
    for k:=1 to part.numChildren loop
      --dont count parts in the entrance buffer
      if part.childNo(k).op > 0 then
        --add the complete working time
        --from the workplan
        WIP:=WIP+workplan[1,part.name].sum(
          {2,1}..{2,*});
    end;
    next;
  next;
  --if enough WIP then move the next part
  if buffer1.occupied and
    (WIP + workplan[1,buffer1.cont.name].sum(
      {2,1}..{2,*})) <= max_WIP and
    M1.empty and M1.operational then

```

```

        buffer1.cont.move (M1) ;
    end;
end;

```

### 4.11.2 Overall System Availability, Line Down Time

The availability of a machine can be taken into account without much effort within the simulation. If actual data are available for evaluation, you can achieve a good representation of the behavior of the station. It becomes more difficult when the availability of a complete system consisting of a number of individual machines and conveyor systems has to be determined or detected.

Important tasks in this context are:

- allocation and dimensioning of buffers
- failure planning (e.g. bypass solutions)
- setup time optimization

Usually, a distinction is made between production conditional failures (e.g. tool breakage, problems with handling technology, quality problems) of limited duration and averages (total failure with long standstill duration). Availability information usually includes only disturbances with certain predictable probabilities. Often, there is a demand that the disturbance (up to a certain duration) of a machine must not lead to a failure of the entire line. To achieve this, buffer or bypass solutions must be integrated into the line. The following simple example will serve to demonstrate.

#### Example: Overall System Availability

Create a frame like in Fig. 4.77:

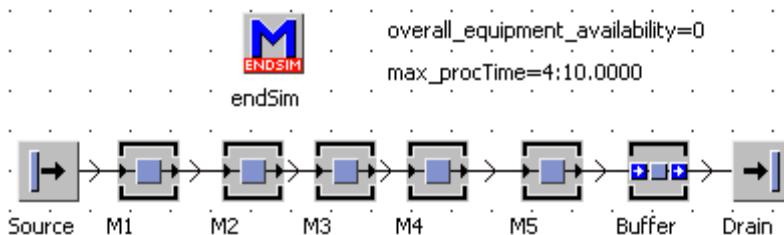


Fig. 4.77 Example Frame

Create the following settings:

Machine	Processing time	Availability	MTTR
M1	3:50	95%	35:00
M2	2:50	96%	20:00
M3	4:10	92%	1:00:00
M4	2:30	98%	35:00
M5	1:00	90%	1:00:00

Use for each machine another random stream (until Plant Simulation 11). The generation interval of the source based on the bottleneck station M3 is 250 seconds / Availability 0.92 = 272 seconds. Let the source generate the parts blocking.

#### *Calculation of Overall System Availability*

Arithmetically, the overall availability of the system is the product of the individual availabilities:  $0.95 * 0.96 * 0.92 * 0.98 * 0.9 = 0.74$

However, this calculation does not include the effects of buffering, logistics delays or quality assurance. In the simulation, we can use other approaches to determine the overall system availability. An approach that is mainly used in control systems is the determination of the line down-time and, from this, the calculation of the overall system availability using the formula:

$$\text{Availability} = (\text{Total Time} - \text{Down Time}) / \text{Total Time}$$

#### *Determination of the Down-Time within the Simulation*

The down-time is recognized statistically by Plant Simulation within the simulation as a waiting time. In a line production, the longest processing time determines the exit distance of the MUs. If you introduce an element with a processing time = maximum processing time at the end of the production line, then the line down times (gaps) will be registered as waiting time in this station (attribute `statWaitingTime`). The station used to measure must be able to deliver the MUs at all times; otherwise, blockages will cause a falsification of the result. The drain is ideal for such a purpose. To prevent a jam in front of the drain, you should use a buffer before the drain position (see example, capacity: -1). The longest processing time in this example is 4:10, set in the drain a processing time of 4:10. Select in the EventController a simulation period of 1,000 days and start the simulation without animation. With the `endSim` method, the overall system availability can be determined according to the following scheme:

```
is
do
  overall_equipment_availability:=
    (eventController.simTime-
     drain.statWaitingTime) /
     eventController.simTime*100;
end;
```

By random spread of failure duration and failure distance, the result (75 per cent) is rarely identical with theoretical calculation (74 per cent). The result you can see directly in the statistics of the drain (see Fig. 4.78 working).

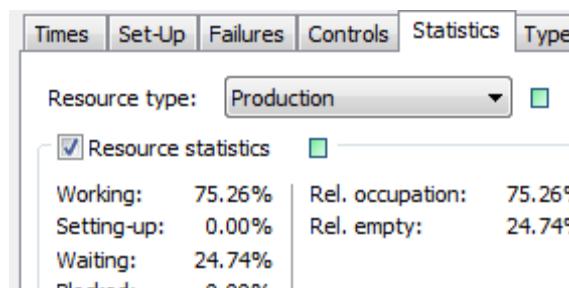


Fig. 4.78 Statistics Drain

#### *Indirect Determination of Overall System Availability*

Another possibility is calculation on the basis of theoretically possible and realized output. The ratio of actual output (drain.statNumIn) and theoretically possible output also gives the overall system availability (method endSim):

```
is
do
  overall_equipment_availability:=
  drain.statNumIn/
  (eventController.simTime/max_procTime)*100;
end;
```

#### *Buffer Allocation and Buffer Dimensioning*

A basic idea is to allow the bottleneck station to work at all times. For this, the station should be decoupled from the disturbances of the predecessor (by a little stock of parts) and able to deliver parts even if the successor is failed. Place each buffer before (buffer1) and after station M3 (buffer2). Set the capacity to -1 (unlimited). In order to observe the tendency of the buffer stock, add two TimeSequence objects and two charts into the frame. The TimeSequences should record the progress of the buffer stocks. To do this, settings like in Fig. 4.79 are necessary:

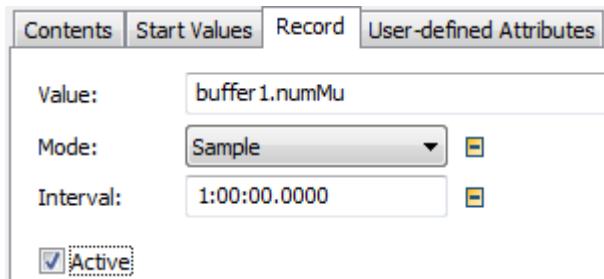
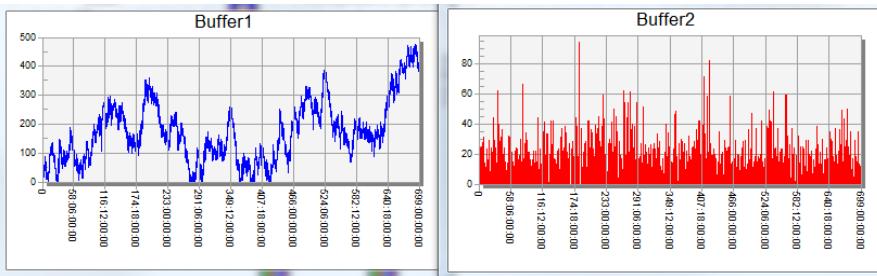


Fig. 4.79 Settings TimeSequence

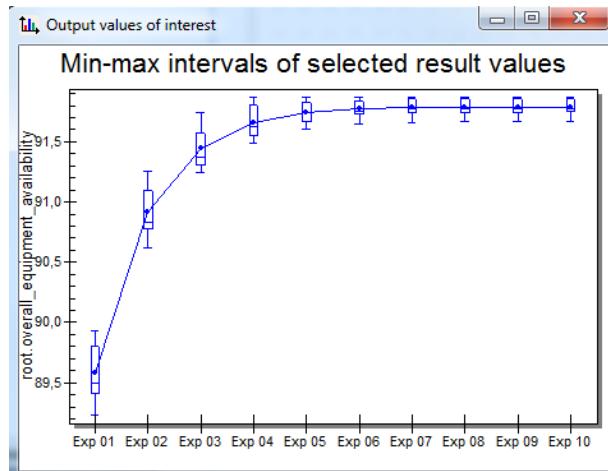
Add two further chart elements in the network, and drag the TimeSequence to each of the charts for display (the Chart block is automatically configured). The profile of the buffer stocks is very different (Fig. 4.80).



**Fig. 4.80** Buffer Stock

Usually, technical and economic considerations play a role in the buffer design. During Buffer2, most scenarios are covered with 40 places; for Buffer1, no statement is possible. In such cases, sensitivity analyses are used. In the simulation, this means that tests are carried out with increasing buffer size and the change in the overall availability is observed (Experiment Manager).

Set the capacity of the second buffer to 50. Insert an ExperimentManager into the frame. The output value is `root.overall_equipment_availability` and the input value is `root.buffer1.capacity`. Define experiments from 10 to 100 (10 steps). The result shows that increasing the buffering capacity to over 30 causes an increasingly smaller rise in overall system availability (Fig. 4.81).



**Fig. 4.81** Result Chart Experiment Manager

Set Buffer1 to a capacity of 30. Test the results. With the help of two buffers, you can increase the overall system availability from 74 per cent to over 91 per cent (this corresponds to an increase in output by more than 20 per cent).

### 4.11.3 Sequence Stability

In many areas, it is necessary to supply a subsequent production with a fixed sequence of parts (e.g. the assembly line of a vehicle manufacturer must be supplied with a precisely defined sequence of vehicles). In fact, turbulences occur in the production (sequence violations). Reasons for this may include, among other things, be time-consuming special production processes, quality control, or quality problems (rework). Within the production, steps must be provided at suitable positions to form the sequence again. Sequence stability within the simulation is an indicator which that has to be calculated dynamically. It appears compressed in one indicator if the process is able to supply the following process in the required sequence of parts.

#### Example: Sequence Stability

A production section is composed of eight stations and a finish line. All parts are first processed on stations M1 to M4. Thereafter, a special processing is planned for part T4; all other parts are processed on M5. After the station M6 (all parts), the parts go through a quality check. 10 per cent of the parts must be reworked at station Q. Create a frame according to Fig. 4.82 Example .

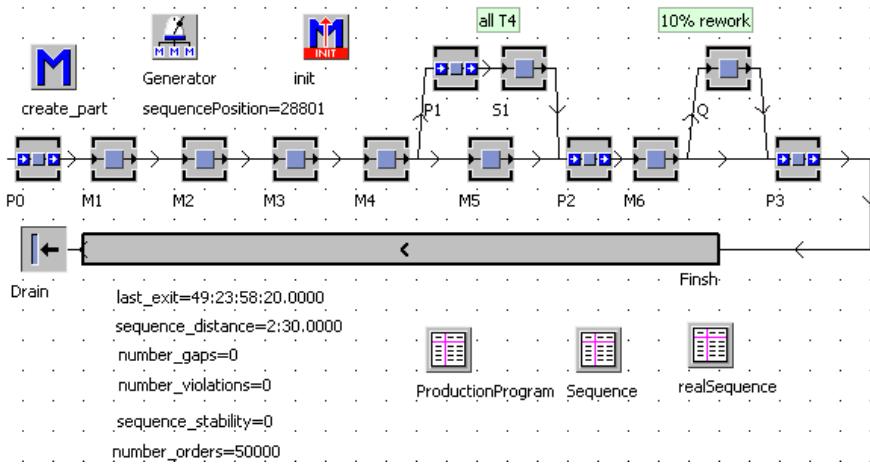


Fig. 4.82 Example Frame

Ensure the following basic settings: The machines M1, M2, M3, M4, M5, and Q have a processing time of 2:30, and M6 of two minutes. S1 has a processing time of 10:00. The conveyor "finish" has a speed of 0.006 m/s and a length of 21 meters. P0, P1 and P2 have a capacity of four; the buffer P3 has a capacity of 20.

The variables have the following data types: sequencePosition: integer, initial value 1; last\_exit: time, initial value 0; sequence\_distance: time, initial value 0; number\_gaps: integer, initial value 0; number\_violations: integer, initial value 0; sequence\_stability: real, initial value 0; number\_orders: integer, value 50,000.

#### *Pearl Chain, Production Sequence*

The first step is the creation of a pearl chain (production sequence). Often, the pearl chain is available as a day call list or protocols from the production control systems. In some cases (e.g. the integration of new products), the pearl chain for the simulation must be created. The basis could be, for example, a planned production program. Create five different entities (T1 to T5). All entities need a user-defined attribute sequence (integer). Create the parts in different colors. Format the table productionProgram according to Fig. 4.83 and enter the data.

	integer 1	real 2	string 3
string	Part_Code	Percentage	Name
1	1	20.00	T1
2	2	25.00	T2
3	3	20.00	T3
4	4	10.00	T4
5	5	25.00	T5

**Fig. 4.83** TableFile ProductionProgram

Starting from the table productionProgram (as a distribution table), you can use the method `z_dEmp(<table>)` to produce a pearl chain. The variable `num_orders` specifies the number of production orders. The sequence is written to the table sequence. Format the table sequence as shown in Fig. 4.84.

	integer 1	string 2
string	Sequence-Number	Part
1		
n		

**Fig. 4.84** TableFile Sequence

We produce the pearl chain in the init method. The init method first deletes the content of the sequence table, and then dices in each loop one part code. Sequence number and the part name are written sequentially in the sequence table:

```
is
  i:integer;
  part_no:integer;
do
```

```

sequence.delete;
realSequence.delete;
for i:=1 to number_orders loop
  part_no:=z_dEmp(1,productionProgram);
  sequence.writeRow(1,sequence.yDim+1,
    i, productionProgram[3,part_no]);
next;
end;

```

According to the pearl chain, the parts must be generated. In the current example, we use a combination of a generator and a method (create\_part). The generator calls the method at intervals of 2:30. The method creates from the part name an object reference and generates one part in the buffer P0. The current sequence number is stored in the part. Thereafter, the sequence is increased. When the end of the sequence table is reached, the simulation is terminated with an error message. Method create\_part:

```

is
  partClass:object;
  part:object;
do
  -- create link
  partClass:=
  str_to_obj(".MUs."+sequence[2,sequencePosition]);
  -- create part
  part:=partClass.create(P0);
  -- save sequence number
  if part /=void then
    part.sequence:=sequencePosition;
    -- increase sequencePosition for the next call
    sequencePosition:=sequencePosition+1;
    -- stop at the end of the sequence table
    if sequencePosition> sequence.yDim then
      messageBox("Last sequence created!",1,13);
      eventController.stop;
    end;
  end;
end;

```

### *Routing*

Select for M4 Exit Strategy: MU attribute; default successor: 2; attribute type: string. The flow of material is to be divided according to the name of the MUs, so that T4 is moved to P1 (e.g. see Fig. 4.85 if P1 is the successor number 1 and the standard successor is 2).

	Attribute	Value	Successor
1	Name	T4	1

Fig. 4.85 Exit Strategy of M4

The exit strategy of M6 is random (10 per cent in the direction of Q, 90 per cent directly to P3).

#### *Sequence Violations and Gaps; Calculation of Sequence Stability*

The outward transfer of a part from the pearl chain can cause at least two effects:

- There may be a gap.

- The part is no longer in its position in the pearl chain (sequence violation). Gaps create costs (unused working time, unused machine time). Each sequence violation causes costs in the material disposition of the downstream processes (e.g. assembly). In the calculation of sequence stability (*RST*), therefore, the number of the gaps (*L*) and the number of sequence violations (*SV*) are included. The sequence stability could be e.g. calculated as follows<sup>2</sup>:

$$RST = \frac{n - \sum_{i=1}^l L_i - \sum_{j=1}^s SV_j}{n}$$

With:  $n$  = total number of sequences

$l$  = number of gaps

$s$  = number of sequence violations

In the simulation, we need to recognize and count both gaps and sequence violations for this purpose. To facilitate the detection of the gaps, set a cycle time of 2:30 in the conveyor finish. This results in the exit distance of the MUs always being an integral multiple of the cycle time. When the exit distance between the MUs is greater than the cycle time, there is a gap. To make the result of the calculation of the sequence stability verifiable, we will enter gaps and sequence violations into a table (realSequence). Format the table as in Fig. 4.86.

	integer 1	string 2	string 3	int 4
string	Sequence-No	Gap	Sequence violation	
1				

Fig. 4.86 Formatting of TableFile realSequence

At the end of the finish conveyor, the sequence stability must be measured (exit control rear). For this, the cycle distance is determined. If the distance is greater than the cycle time of the finish conveyor, then the gap is registered ("X" in the second column of the table realSequenz, add one to the variable number\_gaps). Sequence violations are determined by comparing the value of the sequence

<sup>2</sup> Klug F (2010) Logistikmanagement in der Automobilindustrie. Springer, Heidelberg

attribute of the part with the current sequence position. If the sequence is less than the next sequence position, the sequence is entered into the third column, and the number of sequence violations is increased.

Exit control finish:

```

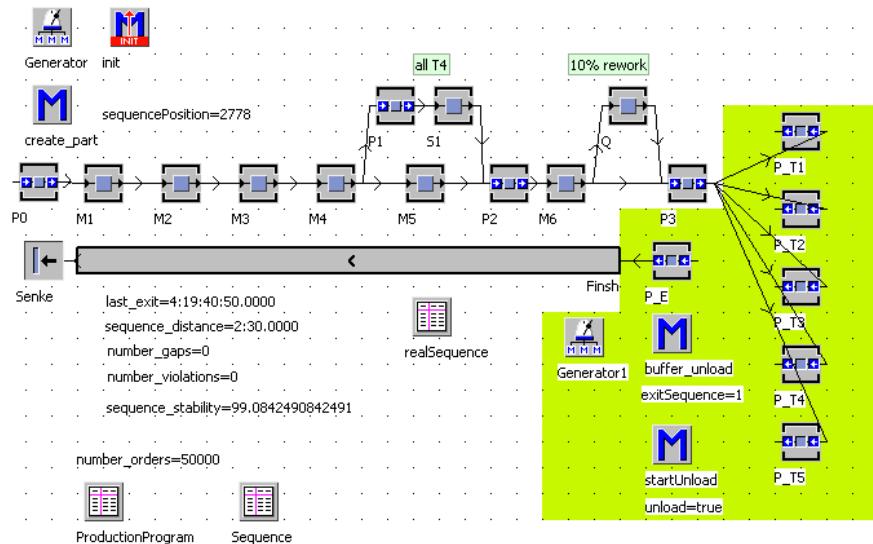
is
do
  if realSequence.yDim > 0 then
    sequence_distance:=
      eventController.simTime-last_exit;
  end;
  if sequence_distance>self.~.cycleTime then
    --gap
    number_gaps:=number_gaps+1;
    realSequence.writeRow(2,realSequence.yDim, "X");
  end;
  if @.sequence = realSequence.yDim+1 then
    realSequence.writeRow(1,
      realSequence.yDim+1,@.sequence);
  elseif @.sequence < realSequence.yDim+1 then
    --too late
    realSequence.writeRow(3,realSequence.yDim,
      realSequence[3,realSequence.yDim]"/"
      +to_str(@.sequence));
    number_violations:=number_violations+1;
  else
    --jump over
    realSequence.writeRow(1,@.sequence,@.sequence);
  end;
  last_exit:=eventController.simTime;
  -- calculate sequence stability
  sequence_stability:=(realSequence.yDim-
    number_gaps-number_violations)/
    realSequence.yDim*100;
end;

```

In the present example, the sequence stability is about 87 per cent.

*Increasing the sequence stability: sequence/sort buffer*

One way to increase the sequence stability is to set up sorting or sequencing buffers. There, the parts can be buffered separated by type and color, and removed according to the sequence order. The buffer size can be determined via simulation runs. Extend the simulation model like in Fig. 4.87 (green area).



**Fig. 4.87** Extended Model

Instead of relocating the parts directly from P3 to finish, the parts are stored and sorted by name in single buffers. From there, the parts will be relocated in the correct sequence on P\_E. The exit strategy of the buffer P3 is MU Attribute (blocking). The MU Attribute is name (String). You must complete the list so that T1 reaches P\_T1 and so on. The buffers P\_T1 to P\_T5 have a capacity of seven. P\_E has a capacity of one. The attribute load has a data type Boolean and false as a start value, with exitSequence as integer and initial value as one. Generator1 calls the method buffer\_unload every 10 seconds.

Preliminary consideration: The sort buffers should have a certain stock of parts before the start of unloading (the first buffer should be full). Only in this way can gaps be bridged. In the example, this is solved as follows. In the init method, a method is called at the end (startUnload). This method is waiting for the first full buffer and sets the variable "unload" to true. The unload method (buffer\_unload) asks the variable "unload" and quits as long as the value of the variable is false. Method startUnload:

```

is
do
  waituntil P_T1.full or P_T2.full or P_T3.full
  or P_T4.full or P_T5.full prio 1;
  unload:=true;
end;

```

Insert a call of the startUnload method in the init method.

The unloading of the sort buffers can be most easily realized with a combination of generator and method (from "outside the buffer" by a central

instance). The method `buffer_unload` first reads the next required part in the sequence from the sequence list, creates a link to the relevant buffer and moves a part if the buffer is occupied. If the relevant buffer is unoccupied (gap), the value of the variable `exitSequence` is not increased and there is a new trial at the next call of the generator. This creates a robust control. Method `buffer_unload`:

```

is
  part:string;
  buffer:object;
do
  if unload then
    --request part from the sequence
    part:=sequence[2,exitSequence];
    buffer:=str_to_obj("P_"+part);
    if buffer.occupied and P_E.empty then
      buffer.cont.move(P_E);
      exitSequence:=exitSequence+1;
    end;
  end;
end;

```

Despite the ordered removal from the sorting buffers, sequence violations may occur. These are parts from the same type that have "overtaken" and are now lying in an incorrect sequence one after the other in the sorting buffer. In many productions, the exact sequence position if the same type is unimportant up to a certain position (for example, it only needs to have the right color or type). One solution is that the control assigns the sequence number to the part after exiting the sequence buffer (so-called late-order binding). In the present example, only a small addition in the method `buffer_unloading` is required (in bold):

```

is
  part:string;
  buffer:object;
do
  if unload then
    --request part from the sequence
    part:=sequence[2,exitSequence];
    buffer:=str_to_obj("P_"+part);
    if buffer.occupied and P_E.empty then
      buffer.cont.move(P_E);
      P_E.cont.sequence:=exitSequence;
      exitSequence:=exitSequence+1;
    end;
  end;
end;

```

Through the combination of sequence buffers and late-order binding, most sequence violations can be prevented in the example.

# Chapter 5

## Working with Random Values

In many cases, it is not possible to specify a fixed value for processing time or failure duration. Plant Simulation provides functions for including random and empirical distributions of values in the simulation.

### 5.1 Working with Distribution Tables

When entering times, you can choose from among the menus of a number of distribution functions. The distribution functions can, in principle, be divided into two areas:

- Empirical distributions: This includes real values—e.g. from the past—and the distribution of values can be specified.
- Probability distributions: Here, the real distribution is not known exactly. It will work with mathematical distribution functions that map the distribution of real values approximately.

A fundamental problem is data collection. You need to estimate the distribution of a large amount of data, which should be collected for a certain period. Especially for new solutions, these data are available only from comparing facilities. In any case, try to reproduce the real fluctuations of values as closely as possible in the simulation in order to achieve a realistic result.

Proceed to determine empirical distributions as follows:

1. Determine the maximum and the minimum of the values (eliminate abnormally high/low values)
2. Divide the data into a suitable number of classes (dependent on the distance from minimum and maximum, and the quantity of the data)
3. Determine the class width (class boundaries) by dividing the difference of maximum value and minimum value with the number of classes
4. Calculate the lower and upper bounds of the classes
5. Count the dates and assign the data into the classes (the class usually includes the upper bound and not the lower bound)
6. Evaluate the frequencies of the various classes by setting the number of values in each class in proportion to the total number of values

## Example: Analysis of Statistical Data

The following example shows the evaluation of statistical data (frequency) using Excel, which has a number of powerful features for evaluating statistical data. You can import, via ASCII, data from different sources. As an example, the following recorded values are to be evaluated: 90, 78, 89, 67, 78, 88, 99, 78, 77, 76, 80, 81, 77, 76, 75, 72, 80, 82, 80, 91, 92, 94. Normally, the data are available in table form or as a comma-delimited text file (.csv, .txt, .asc, etc.). First, enter the data into an Excel table in one column. In the first step, determine the maximum and the minimum of the values. For this, you can use the Excel functions `max(range)` and `min(range)`. In this example, the maximum is 99 and the minimum 67. The difference between the two values is 32. We must now divide this value into a sufficient number of classes. The more classes you create, the higher is the accuracy of the mapping of the variation. We divide the total range into eight classes with a class width of four (division of distance by class number). As a next step, we define the upper and lower limits of the classes (lower limit + distance = upper limit = lower limit of the next class). Fig. 5.1 shows how this could be carried out in Excel.

max	99	
min	67	
difference	32	
num. classes	8	
class width	4	
	lower limit	upper limit
class1	0	71
class2	71	75
class3	75	79
class4	79	83
class5	83	87
class6	87	91
class7	91	95
class8	95	99

**Fig. 5.1** Classes in Excel

In the next step, we determine the frequency of the individual values in the data. To do this, we need a matrix function of Excel. These functions work only when you perform a specific series of steps:

1. Select the range in which you want to display the frequencies (cells near the upper limits)
2. Press the F2 button (the cursor starts blinking in the first cell)
3. Enter the following formula: "= Frequency" (mark the area with the data; if the area is larger, then select the first cell and press the key combination Ctrl + Shift + directional button arrow down—" ;"—Mark the cells with the upper limits—Close the parenthesis ")")
4. Finally, press Ctrl + Shift + Enter

Note: The name of the function depends on the language of your Microsoft Office installation. "Frequency" works only for English installations.

Excel determines relatively quickly the frequency of the data within the class limits. As a last step, we have to calculate from the absolute frequencies the relative frequencies. Hence, we set in ratio the individual absolute frequencies to the number of values (absolute value/sum of all absolute frequencies, Fig. 5.2).

num. classes	8				
class width	4				
	lower limit	upper limit	frequency	%	
class1	0	71	1	5%	
class2	71	75	2	9%	
class3	75	79	7	32%	
class4	79	83	5	23%	
class5	83	87	0	0%	
class6	87	91	4	18%	
class7	91	95	2	9%	
class8	95	99	1	5%	

**Fig. 5.2** Excel Frequencies

This value distribution can be used in Plant Simulation.

Plant Simulation provides several features to take into account empirical distributions. There are three ways to work with empirical data:

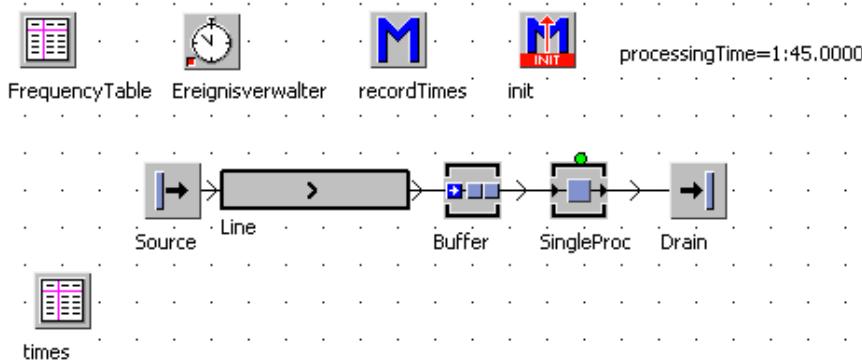
- Emp
- cEmp
- dEmp

#### **Example: Empirical Distributions with dEmp**

In a work process, a continuous process with a 1:40 cycle time is followed by a manual process. In the manual process, the worker must measure the part and do rework depending on the result. The process time of the workplace is distributed as follows:

Process time	Percentage of all measurements
1:10–1:20	5%
1:20–1:30	15%
1:30–1:40	60%
1:40–1:50	20%

There is a buffer for decoupling the manual workstation from the previous process. Create a frame according to Fig. 5.3 Example Frame.



**Fig. 5.3** Example Frame

The source generates parts with an interval of 1:40. The buffer has a capacity of four and a processing time of zero seconds. The processing time of the SingleProc is based on a distribution table. Choose dEmp as distribution in the field Processing time of the SingleProc, and enter Stream 1 and the table FrequencyTable separated by commas (Fig. 5.4). Starting from Version 11, you no longer need to specify a random stream.



**Fig. 5.4** Setting Processing Time, dEmp Plant Simulation Version 10.1

When you click Apply, the table will be formatted. Enter the distribution as follows: As the value for the processing time, enter the middle of the value range and the corresponding frequency (e.g. in the range from 1:20 to 1:30, the middle value is 1:25). The filled in FrequencyTable should resemble Fig. 5.5.

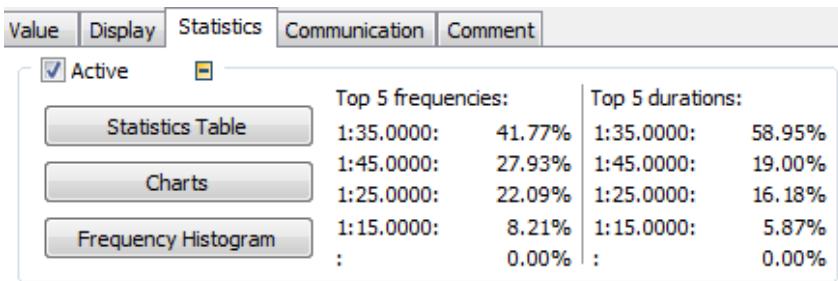
	time 1	real 2
string	ProcTime	Frequency
1	1:15.0000	5.00
2	1:25.0000	15.00
3	1:35.0000	60.00
4	1:45.0000	20.00

**Fig. 5.5** FrequencyTable

When you start the simulation, Plant Simulation dices a new processing time before a new part is moved to the SingleProc. As it is hard to observe this behavior, we can apply a little trick here. The global variable is able to collect statistics. Thereby, the global variable records the frequency and duration of the values of the variable (which works only with the data type string and integer). For this, we need to write a short method. To do this, open the method recordTimes by double-clicking. Enter the following text into the method:

```
is
do
  processingTime:=time_to_str(??.procTime);
  @.move;
end;
```

If you cannot enter text, deselect the option "inherit source code" (to the left of the green check mark). With this statement, we store the processing time of SingleProc in the variable processingTime as text. To ensure that the method is called often enough, we assign it to the exit control (front) of the SingleProc. We need to enable the statistics of the global variable. Open the global variable by double-clicking. Select in the tab Value the data type string and activate in the Statistics tab the statistics by activating the checkbox Active. If you now run the simulation for a while, you will see the distribution of the values in the Statistics tab. The duration of the values should reflect the frequencies entered in the table FrequencyTable (Fig. 5.6).

**Fig. 5.6** Statistics Global Variable

With Emp, you can use a primitive empirical distribution in your model. The frequency table consists of only one column. For each row (value), you can specify a frequency. Plant Simulation interprets in this distribution, e.g. in relation to the processing time, the line number as seconds.

### Example: Emp

Use the previous example. Change the generation interval of the source to three seconds. Add an additional TableFile into the frame (Table1). In the SingleProc, choose the distribution of the processing time as in Fig. 5.7.



Fig. 5.7 Setting Emp (Plant Simulation 10.1)

The first parameter sets the used random stream (each random distribution must have its own stream only before Version 11). Next is the table with the distributions and the third parameter is the column within the table that contains the distributions (by default, the first column). You must include the following distribution: 20 per cent processing time for one second, 30 per cent for two seconds, 40 per cent for three seconds and 10 per cent for four seconds. The necessary entry in Table 1 is shown in Fig. 5.8.

	real
	1
string	Frequency
1	20.00
2	30.00
3	40.00
4	10.00

Fig. 5.8 Emp Distribution

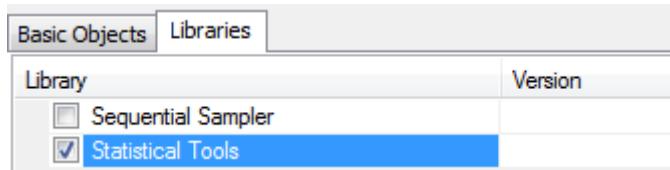
With cEmp, you can consider distributions with their class boundaries. The values are specified each with their lower and upper class limits. In this way, you can take the data directly from statistical analyses (distribution functions). In the example of dEmp, the distribution table would appear as in Fig. 5.9.

	time 1	time 2	real 3
String	left Border	right Border	Frequency
1	1:10.0000	1:20.0000	5.00
2	1:20.0000	1:30.0000	15.00
3	1:30.0000	1:40.0000	60.00
4	1:40.0000	1:50.0000	20.00

**Fig. 5.9** cEmp Distribution

## 5.2 Working with Probability Distributions

If you have a lot of data with a relatively large variation, then an empirical evaluation may be too inaccurate or complex. As an alternative, it is possible to use a probability function to represent the variation of the data. Plant Simulation provides for this purpose a number of functions. The biggest problem is to find an appropriate function for representing the "behavior" of the data. To find the right function, you can use the package DataFit within Plant Simulation. For this, you must add the statistical tools to your class library. Select File—Manage Class Library. Click in the following dialog on the tab Libraries. Activate in the area Tools the statistical tools (Fig. 5.10).



**Fig. 5.10** Manage Library

Now you find the statistical tools in the Tools folder and in the toolbox.

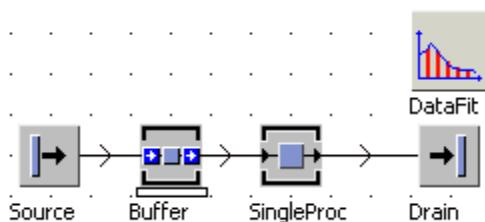
### 5.2.1 Use of DataFit to Determine Probability Distributions

If you have a larger number of data (> 60 records), use DataFit to search a probability distribution that describes the data with sufficient accuracy. You can enter data directly into DataFit or import them from a text file. The data file must be separated by a fixed separator.

#### Example: DataFit

You should simulate a SingleProc. The processing time of the SingleProc fluctuates at around 4.62 seconds. The values are stored in a file distrib.csv.

Use DataFit to determine the appropriate distribution for the processing time. Create a frame as in Fig. 5.11.

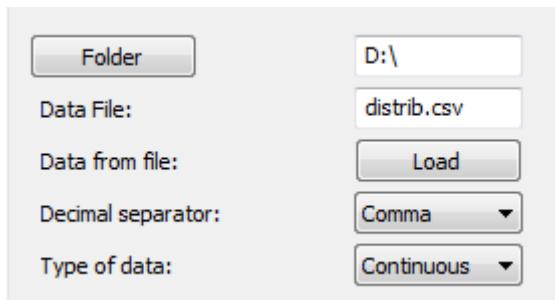


**Fig. 5.11** Example Frame

The source generates one part in 4.62 seconds. The buffer has a capacity of four places and no processing time. The drain has a processing time of zero seconds. The data can be downloaded from [www.bangsow.de](http://www.bangsow.de); look for the example DataFit. Open the DataFit object by double-clicking. Click on Folder and select the folder where the file with the data is located. Enter the name of the file in the field Data File. Define the decimal point (in the case of distrib.csv comma). In the Type of data field, there are two choices:

- discrete; the data are integers and greater than zero
- continuous; the data can also contain negative values or floating point numbers

In the present example, the data consists of positive floating point numbers. So, you need to select Continuous. The filled-in Input of Data tab of the DataFit object should look like Fig. 5.12.



**Fig. 5.12** DataFit Data Input

Click first on Apply and then on Load. When all the settings are correct, the data are loaded into the data table of DataFit. If you are using DataFit for several analyses, you must first remove the old data by clicking the Delete button. With the Open button, you can look at the imported data (you can also import data

directly, insert via the clipboard or enter). In the next step, clean up the data. When recording the data, "outliers" may occur—i.e. data that deviate extremely from the remaining values. This can happen, for example, if the processing time is determined only by recording the start and end of the operation. If the system is disturbed and the part can be finished only after repair, then this single time must be deleted in the statistical series. Remove the outliers from the value table. In the tab Fit you can now allow DataFit to look for a distribution. Set a level of significance (between 0.0 and 0.05). This level usually indicates the per cent of the values that may lie outside the expected distribution. Insert the number of classes (follow if possible the recommendations of the program). Click on FIT to search for suitable distributions. With Histogram—Show you can display a histogram with the set number of classes (Fig. 5.13).

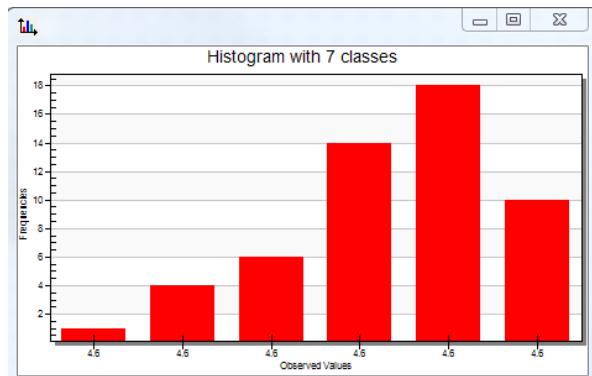


Fig. 5.13 DataFit Histogram

The result of the examination is visible in the tab Evaluation by clicking on Results—Open. Plant Simulation uses several methods to assess the suitability of a specific distribution for the data. The order in the table Ranking represents the suitability of the distribution. In our example, the best solution is a normal distribution. You can find the parameters to be set in Plant Simulation in the last columns of the Ranking table (Fig. 5.14).

	string 0	real 10	real 11	real 12	string 13	string 14	string 15
string	Distribution	Parameter1	Parameter2	Parameter3	Parameter 1	Parameter 2	Parameter 3
1	Normal	4.6187	0.016803075		Mu = 4.6	Sigma = 0.017	
2	Triangle	4.62	4.574	4.651	c = 4.6	a = 4.6	b = 4.7
3	Uniform	4.574	4.651		Start = 4.6	Stop = 4.7	
4	Negexp	4.6187			Beta = 4.6		

Fig. 5.14 Distribution Parameters

The settings needed to ensure that the processing time of the SingleProc matches DataFit are as follows: distribution: normal; MU = 4.6; sigma = 0.017. Add the definition of a random stream (before Version 11) as well as the lower and upper bounds (mainly to avoid negative values). This resembles Fig. 5.15.



**Fig. 5.15** Process Time, Normal Distribution

### 5.2.2 Use of Uniform Distributions

Some distributions cannot be represented by functions. These are especially values that spread in a completely arbitrary manner within a certain area. When analyzing such values, DataFit cannot determine a distribution function. In such cases, you can instruct Plant Simulation to "dice" the values between a minimum value and a maximum value. The individual values then randomly spread between the minimum and maximum values. If, for example, the processing times spread in a completely arbitrary way between one and three minutes, you will need to set the processing time as shown in Fig. 5.16.



**Fig. 5.16** Uniform Distribution

### 5.2.3 Set of Random Distributed Values Using SimTalk

You can also set randomly distributed values (e.g. randomly distributed processing time) via SimTalk.

#### Example: Set Random Values via SimTalk

Change the processing time on a machine depending on the time of day to reflect the influences of night work. Create a frame according to Fig. 5.17.

Assign the method `EntranceControl` as entrance control of the SingleProc (activate before actions). Insert into the shift calendar a third shift (Shift-3) from 22:00 to 6:00. The processing time of the SingleProc should vary in the first and second shifts between 90 and 120 seconds and in the third shift from 110 to 140 seconds. For the distribution of the processing time, use a uniform distribution. First, set a uniformly distributed processing time in the SingleProc as shown in Fig. 5.18.

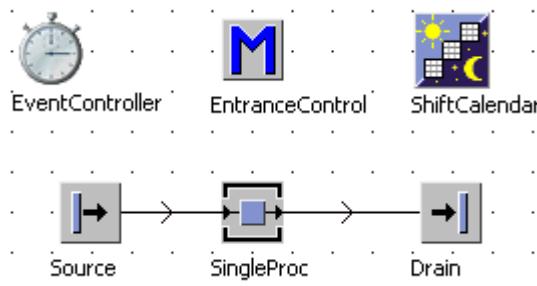


Fig. 5.17 Example Frame



Fig. 5.18 Processing Time

By assigning the time depending on the shift, you can solve as follows: In the entrance control (before actions), query the current shift (`getCurrShift`). Depending on the shift, set the processing time. Thereby, you can access all the elements of the distribution. The notation is `<path>.procTime.<distribution_attribute>`. The names of the distribution elements correspond to those displayed above the input field of the processing time. In the example, set the start and stop values of the uniform distribution. The EntranceControl could appear as follows:

```

is
do
  if ShiftCalendar.getCurrShift = "Shift-3" then
    SingleProc.procTime.start:=num_to_time(110);
    SingleProc.procTime.stop:=num_to_time(140);
  else
    SingleProc.procTime.start:=num_to_time(90);
    SingleProc.procTime.stop:=num_to_time(120);
  end;
end;
  
```

### 5.3 Warm-Up Time

If you use random values, the simulation takes some time to deliver stable statistical values (warm-up time). In addition, you should skip the period from the statistical observation in which the simulation provides no output (no MUs are destroyed in the drain). Use a simple experiment to estimate the warm-up time for your model.

### Example: Warm-up time

Create a frame as in Fig. 5.19.

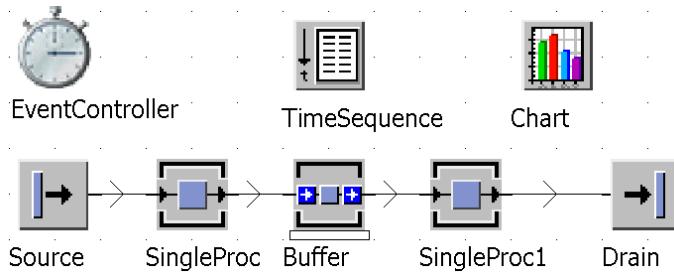


Fig. 5.19 Example Frame

Ensure the following settings: SingleProc: availability of 95 per cent; MTTR: one hour; Buffer: 10 places; SingleProc1: availability of 99 per cent, MTTR of one day. TimeSequence records the attribute Drain.statThroughputPerDay with an interval of four hours. The chart shows the content of the TimeSequence as an x-y graph (line). Let the simulation run for a while. The warm-up phase of your simulation is over when the graph starts to show a horizontal course (Fig. 5.20).

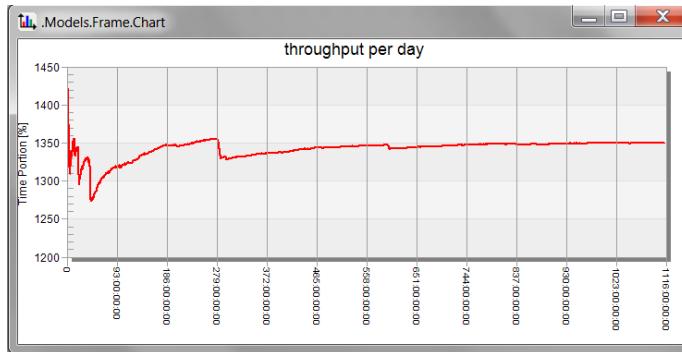


Fig. 5.20 Chart

## 5.4 The ExperimentManager

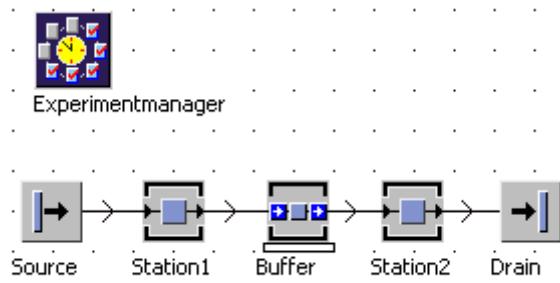
After you have built a simulation model, you will usually perform a series of experiments with it. Experiments generally comprise changes in input variables and the investigation of the effect this has on certain output variables.

### 5.4.1 Simple Experiments

Simple experiments consist of the variation of one input variable; more complex experiments change multiple input variables. In order to ensure that all system states are mapped and simulated, thoroughly plan any experiments that involve several input variables. The ExperimentManager is an important tool for planning and executing experiments. It helps define variable values and result values for observation. The functioning is relatively simple. You define the variation steps of the input variables and the number of simulation runs that then performs Plant Simulation independently. The results of the experiment runs will be recorded and made available for further analysis. You can find the experiment manager in the class library in the Tools folder.

#### Example: ExperimentManager

Determine the optimum size of a buffer with a number of experiments, by investigating the relation between buffer size and output quantity. Simulate two SingleProcs, including one with randomly distributed processing times. For decoupling, a buffer between the SingleProcs is provided. The buffer size should be determined by a series of experiments. Create a frame as shown in Fig. 5.21.



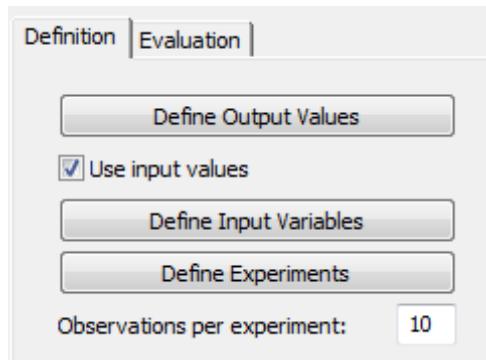
**Fig. 5.21** Example Frame

Create the following settings: The source produces one part (non-blocking) every two minutes. Station1 has a processing time of two minutes. The buffer has an initial capacity of one and no processing time. The processing time of Station2 is uniformly distributed between 0:30 and 3:30 (Fig. 5.22).

Processing time:

**Fig. 5.22** Processing Time

Set the duration for the simulation runs in the EventController in the page settings. Enter in the field End 10 days (10:00:00:00). Open the ExperimentManager by double-clicking (Fig. 5.23).



**Fig. 5.23** Dialog ExperimentManager Definition

First, define the observed value. The output within the 10 days is most easily determined using the statistics of the drain. The relevant attribute is statNumIn. Click the button Define Output Values. Enter into the table root.drain.statNumIn (Fig. 5.24).

	Output Values	Description
1	root.drain.statNumIn	

**Fig. 5.24** Output Value

Next, define the experiments. Activate the checkbox Use Input Values and click Define Input Variables. We want to increase step-by-step the capacity of the buffer in order to investigate the effect of buffer size on the output. The corresponding attribute of the buffer is capacity (If you do not know what attribute you require, right-click on the object and choose from the context menu "Show attributes and methods," select an entry in the list and press F1 to get help on this topic). Add root.buffer.capacity into the table Input Variables (Fig. 5.25).

	Input Values	Description
1	root.buffer.capacity	

**Fig. 5.25** Input Variables

Confirm with OK and then click Apply in the ExperimentManager dialog. Plant Simulation then formats a table for defining the individual experiments. Click on Define Experiments. Define the experiments according to Fig. 5.26 (insert values until 32).

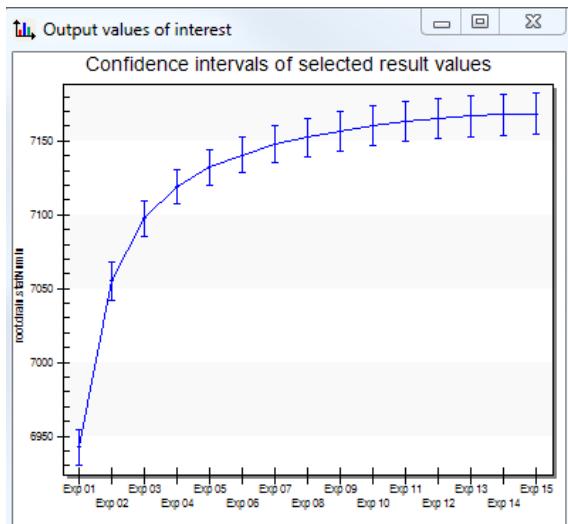
	Active	root.buffer.capacity
1	true	2
2	true	4
3	true	6

**Fig. 5.26** Experiments

Plant Simulation automatically changes the capacity of the buffer before starting the experiments and executes the set number of observations for each experiment. In the EventController, select Tools—Increment variant on reset to run the experiment replications with different random number variants.

Before you start the experiments, ensure that "Show summary report" is disabled in the EventController.

To start the experiments, first click Reset in the ExperimentManager to delete the old values and then click Start. Plant Simulation displays a message box when the experiments are completed. You will find the results in the tab Evaluation (e.g. as diagram, Fig. 5.27).

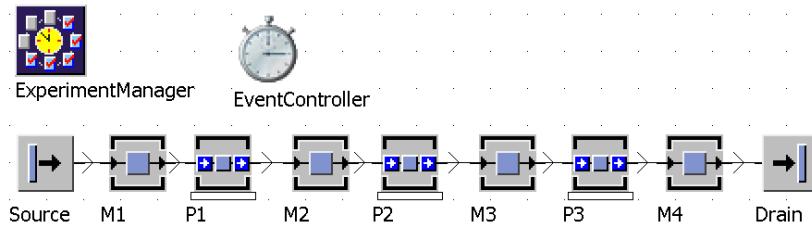
**Fig. 5.27** Experiment Manager Results

#### 5.4.2 *Multi-level Experimental Design*

If you want to change two or more input variables in the experiments, it is called a multi-level experimental design. The problem with such experiments is often the quantity of experiments to be carried out in order to test all the possible value combinations. Plant Simulation helps to generate all the experiments.

### Example: Multi-level experimental design

In the following example, the dimensioning of buffers should be tested with the ExperimentManager. You should investigate a chain of machines. Through the placement of buffers, the output should be increased. Create a frame as in Fig. 5.28.



**Fig. 5.28** Example Frame

Create the following settings:

M1: Processing time 15 seconds

M2: Processing time 60 seconds, 95 per cent availability, 25 minutes MTTR, tool change in interval of two hours working time, duration 25 minutes

M3: Processing time 60 seconds, 90 per cent availability, 30 minutes MTTR

M4: Processing time uniformly distributes between 10 seconds and 110 seconds, 95 per cent availability, two hours MTTR

We are looking for the optimal size of the buffers P1, P2 and P3. In the first step, the output should be increased. Therefore, you should use the ExperimentManager and follow these steps:

1. Open the ExperimentManager and define Drain.statNumIn as an output value.
2. Define the following input data:
  - root.P1.capacity
  - root.P2.capacity
  - root.P3.capacity
3. Select in the ExperimentManager: Tools—Multi-level experimental design. Define for each buffer the range of values for the experiments (lower level, upper level) and the increment (Fig. 5.29).

	Input value	root.P1.capacity	root.P2.capacity	root.P3.capacity
1	Lower level	50	50	50
2	Upper level	150	150	150
3	Increment	25	25	25

**Fig. 5.29** Multi-level Experimental Design

For three input values and five steps, Plant Simulation creates 125 (5x5x5) simulation experiments. Now, you can execute the experiments.

## 5.5 Generic Algorithms

Multi-level experiments can quickly lead to a huge number of simulation runs. By the application of generic algorithms (GA), the absolute number of experiments to be carried out is reduced considerably. The GA generates on the basis of a simulation result (value of an optimization function) and an optimization direction (minimization or maximization) new experiments step-wise.

Three types of optimization problems can be solved using GA:

- Sequence tasks
- Selection tasks
- Allocation tasks

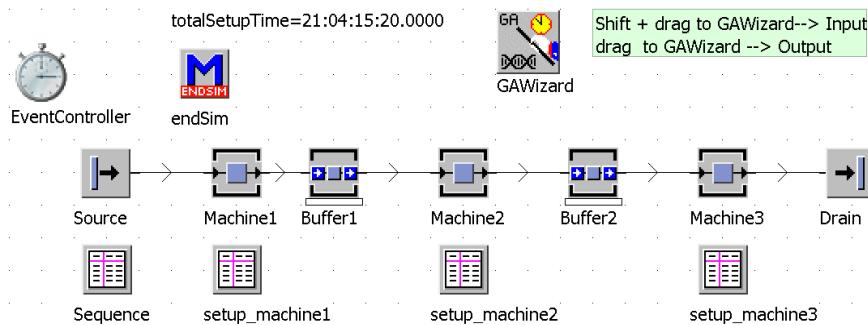
For a GA optimization, you must first create a simulation model. For the simulation model, you have to define a fitness value.

### 5.5.1 GA Sequence Tasks

In sequence tasks, a sequence is varied (stored in a list) for each experiment to determine an optimum.

#### Example: GA Sequence Optimization

A company produces five different products in a pearl chain (lot size 1). We are looking for a sequence with minimum setup time. Create a frame as in Fig. 5.30.



**Fig. 5.30** Example Frame

The source produces MUs according to the table Sequence (sequence cyclical). Fill the table with data from Fig. 5.31.

	object 1	integer 2	string 3
string	MU	Number	Name
1	.MUs.Entity	1	A
2	.MUs.Entity	1	B
3	.MUs.Entity	1	C
4	.MUs.Entity	1	D
5	.MUs.Entity	1	E

**Fig. 5.31** Data Table Sequence

All machines have a processing time of one minute. All machines have for their setup time a "matrix(type)" setting. Assign the relevant tables. The tables contain the times for the setup from one MU type to the next. For Machine1, the setup time matrix might be as in Fig. 5.32. Create setup time matrices for all machines and vary the values.

	string 0	time 1	time 2	time 3	time 4	time 5	time 6	!
string	-	A	B	C	D	E		
1	-	0.0000	1:00.0000	1:00.0000	1:00.0000	1:00.0000	1:00.0000	
2	A	1:00.0000	0.0000	1:00.0000	2:00.0000	5:00.0000	5:00.0000	
3	B	1:00.0000	1:00.0000	0.0000	3:00.0000	5:00.0000	5:00.0000	
4	C	1:00.0000	2:00.0000	3:00.0000	0.0000	10:00.0000	10:00.0000	
5	D	1:00.0000	5:00.0000	5:00.0000	10:00.0000	0.0000	20:00.0000	
6	E	1:00.0000	5:00.0000	5:00.0000	5:00.0000	20:00.0000	0.0000	

**Fig. 5.32** Setup Time Matrix

Set the EventController to an end of 10 days and uncheck the "Show Summary Report" option. Test your model.

### Optimization Function—Fitness Value

Calculate a value for the model by which you are able to assess the quality of a solution. In this example, the entire setup time should be minimized. For each simulation run, calculate the total setup time and make it available (e.g. via a global variable). This works quite well with the help of an endSim method:

```
is
do
  totalSetupTime:=machine1.statSetupTime+
    machine2.statSetupTime+
    machine3.statSetupTime;
end;
```

Add a GAWizard object into your model. Hold down the Shift key and drag the sequence table onto the GAWizard. The GAWizard analyzes the table as an optimization problem. Drag the global variable total setup time onto the GAWizard. The value of the global variable is then automatically used as the fitness value. Open the GAWizard. Select the optimization direction as minimum. The formulation of the optimization problem can be found in the optimization parameter (Fig. 5.33).

	string 1	object 2
string	Parameter:	root.Sequence
1	Sequence of	root.Sequence
2	5 Elements	
-		

**Fig. 5.33** Optimization Parameter

To start the optimization, click Start in the tab Run. After the optimization, a range of evaluations are available (tab Evaluate). Click on Best individuals—Show (Fig. 5.34).

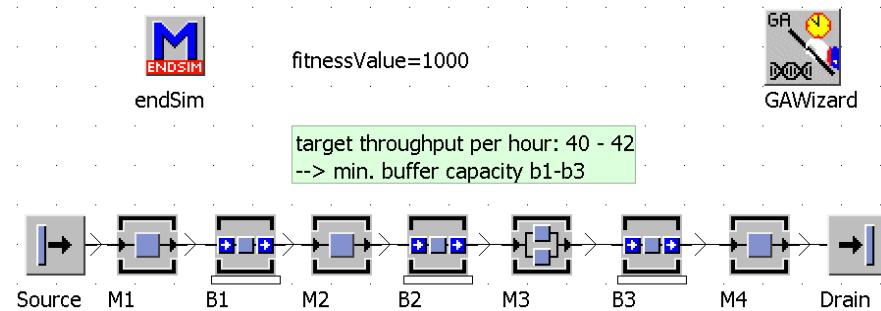
	table 0	time 1	integer 2	table 3
string	Individual	Fitness	ID	Chromosomes -
1	Gen 1 Ind 9	14:19:08:00.00	9	Chrom 1
2	Gen 1 Ind 4	15:16:05:00.00	4	Chrom 1
3	Gen 2 Ind 6	15:16:05:00.00	16	Chrom 1
4	Gen 2 Ind 13	15:16:05:00.00	23	Chrom 1
5	Gen 3 Ind 5	15:16:14:00.00	35	Chrom 1

**Fig. 5.34** GA Results

Double-click for the results in Column 3 in a cell (or click in the cell and press F2). After completion of the optimization, the optimal sequence is set in the table Sequence.

### 5.5.2 GA Range Allocation

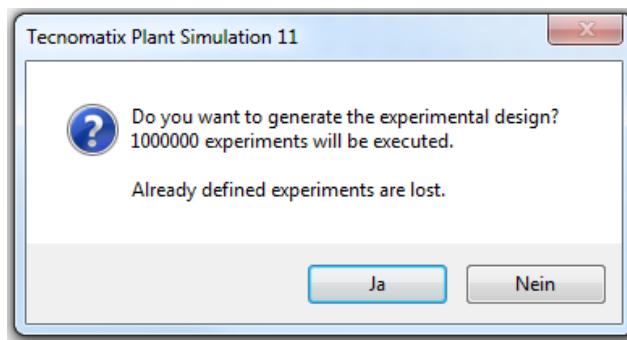
In the range allocation tasks, assign numerical values between minimum and maximum. A classic example is buffer dimensioning. Create a frame as in Fig. 5.35.



**Fig. 5.35** Example Frame

Create the following settings: M1, M2 and M4 each with one minute processing time, 90 per cent availability, 25 minutes MTTR; M3 with four minutes processing time, five places, 60 per cent availability, one hour MTTR. Set the end of the EventController to 50 days and activate the option Tools—Increment variant at reset.

We are looking for the minimum capacity for B1, B2 and B3 at an output of 40–45 parts per hour. One way would be a multi-level experimental design. You could increase the capacity for each buffer step by step from one to 100. Since you work in the simulation with random values (availability), 10 runs should be carried out for each experiment. This would result in 1,000,000 experiments (100x100x100) and 10,000,000 experiment runs. Plant Simulation offers a relevant note if you want to recreate the experiments (Fig. 5.36).



**Fig. 5.36** Plant Simulation Note

Generic algorithms create a useful result with significantly fewer experiments.

### Fitness Value

For the evaluation of the results, the result is considered "not valid" when the output per hour is less than 40 or more than 45. For these cases, you should assign an unfavorable value to the fitness value. In this example, you test capacities up to

100 per buffer (300 in total). The buffer capacity should be minimized. If you set a value of 1,000 in the case of  $<40$  and  $>45$ , then these cases have a disadvantage in evaluating the minimum. The calculation of the fitness value can be carried out in the EndSim method and might look like this:

```

is
do
  if drain.statThroughputPerHour < 40 or
      drain.statThroughputPerHour > 45 then
    fitnessValue:=1000;
  else
    fitnessValue:=b1.capacity+b2.capacity+
      b3.capacity;
  end;
end;

```

Hold down the Shift key, and drag the buffers B1, B2 and B3 onto the GAWizard (optimization values). Select “Capacity” as the attribute. Then drag the global variable fitnessValue to the GAWizard. Open the GAWizard. Select Optimization parameter—Open. Complete the table as in Fig. 5.37 (upper bound, increment).

	string 1	integer 2	string 3	integer 4	string 5	integer 6
string	Parameter:	root.B2.Capacity	Parameter:	root.B1.Capacity	Parameter:	root.B3.Capacity
1	Lower bound	1	Lower bound	1	Lower bound	1
2	Upper bound	100	Upper bound	100	Upper bound	100
3	Increment	1	Increment	1	Increment	1

**Fig. 5.37** Optimization Parameter

Check the fitness calculation (in the table, root.fitnessValue should be entered). Optimization direction is minimum and the number of observations per individual is 10. Start the optimization with Run—Start.

At the end of the optimization, you will find the results in Evaluate—Best Individuals. Double-click on a chromosome (Column 3, Fig. 5.38).

	string 0	string 1
string	Define Set	Allocation
1	root.B2.Capacity	82
2	root.B1.Capacity	25
3	root.B3.Capacity	80

**Fig. 5.38** GA Result

# Chapter 6

## Simulation of Transport Processes

For the modeling of transport processes, Plant Simulation provides e.g. the blocks line, track and transporter. The transporter offers through its configurability a huge number of applications.

### 6.1 The Line

The line is an active material flow object. It transports movable units (MUs) along a route at a constant speed (via an accumulating conveyor like a gravity-roller conveyor or a chain conveyor). MUs cannot pass each other on the line. Unless you have entered an output control or have chosen a different behavior, the line distributes MUs to its successors. When an MU cannot exit (e.g. due to occupation of the successor), the setting "Accumulating" determines whether the MUs maintain their distance or move up.

#### 6.1.1 *Attributes of the Line*

You can find the basic settings in the tab named Attributes (Fig. 6.1).

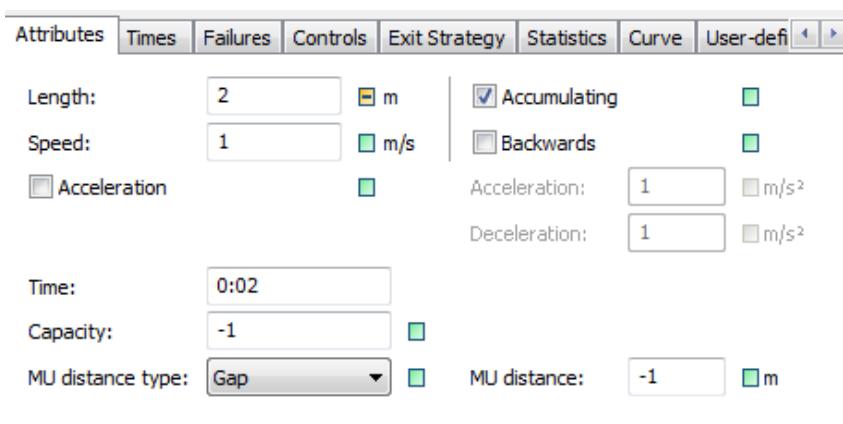


Fig. 6.1 Settings of line

**Length:** Length of the line (the maximum number of MUs on the line is calculated by dividing the length of the line by the length of the MUS).

**Speed:** The line has the same speed along the entire length. You can set the speed to zero to stop the line.

**Time:** Enter the time, an MU needed for transportation from the beginning until the end of the line (the speed is calculated thereby).

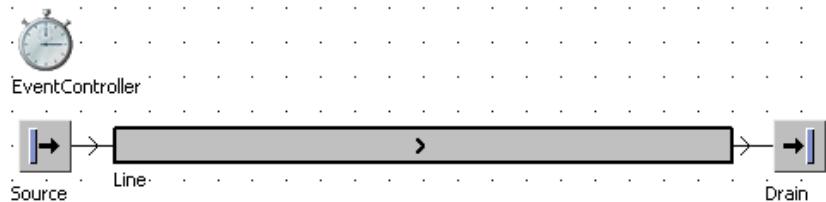
**Capacity:** The capacity determines the maximum number of MUs that can be positioned entirely or in part on the line (-1 for unlimited capacity).

**Backwards:** The line can move forward or backward. Should it move forward, the checkbox Backwards is cleared.

**Accumulating:** See the following example.

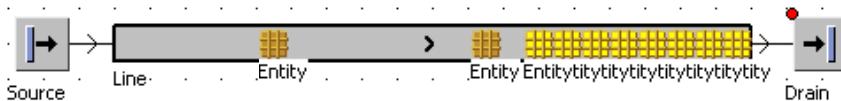
**Example: Line 1:**

Create a frame according to Fig. 6.2. The Source creates each six seconds one part. Prepare the following settings: line: length of 18 meters, speed of one m/s; drain: processing time of zero seconds



**Fig. 6.2** Example frame

The default setting of the line is Accumulating (a checkmark in the box). Now, fail the drain (checkbox Failed) and save your changes. Start the simulation. The MUs move up. Thus, the line works like a buffer. The simulation stops only when the entire line is occupied by MUs (Fig. 6.3).



**Fig. 6.3** Line accumulating

Remove the MUs from the simulation model, clear the checkbox Accumulating in the dialog of the line and confirm your changes by clicking OK. Now restart the simulation. The parts on the line keep their distance (Fig. 6.4). The line cannot be used as a buffer in this setting.

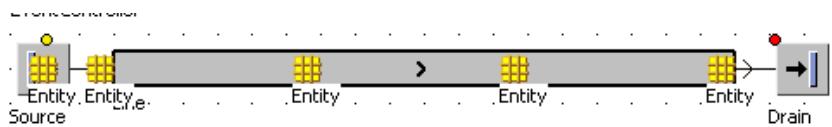


Fig. 6.4 Line not accumulating

The kind of behavior that the line must have depends on the technical realization of the line. Conveyor belts or roller conveyors are normally accumulating; a chain conveyor is generally non-accumulating.

### 6.1.2 Curves and Corners

Lines may have a highly complex course. Plant Simulation allows you to design the course with as much complexity as is present in the real layout. If you have inserted a line in a frame, you can extend it by dragging. You can change the shape of the line using the context menu command Append Points. Plant Simulation draws the length of the line using a setting ratio of meters to pixels. In the basic setting, the grid spacing of a frame is 20 x 20 pixels. You can define a different grid space in the Plant Simulation window under Tools – Preferences – Modeling (Fig. 6.5).

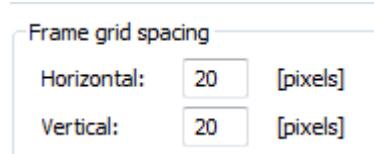


Fig. 6.5 Grid spacing

You can adjust the ratio of grid and dimensions for each frame individually. Select Tools – Scaling Factor in the frame window (Fig. 6.6).

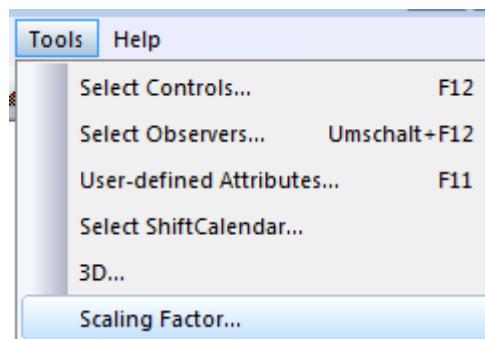


Fig. 6.6 Menu entry for the scaling factor

Enter the required size ratio into the scaling factor dialog (Fig. 6.7).

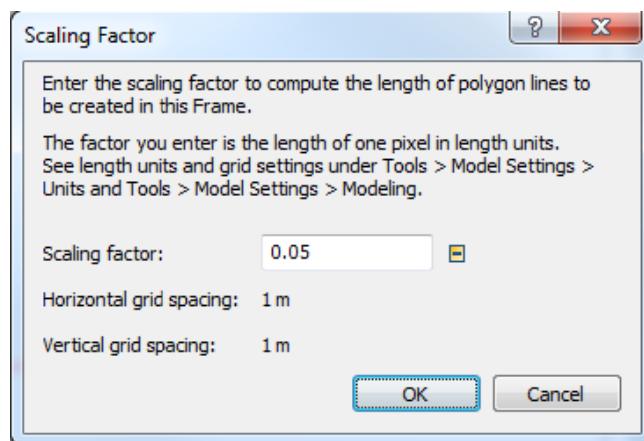


Fig. 6.7 Menu for the scaling factor

Define the visual appearance of the curved object in the tab Curve. Clear the checkbox Active if you want to use a separate icon for the line (e.g. from an icon library). If you append points (right mouse button – Append Points) and hold Ctrl + Shift, you can draw arc segments (Fig. 6.8).

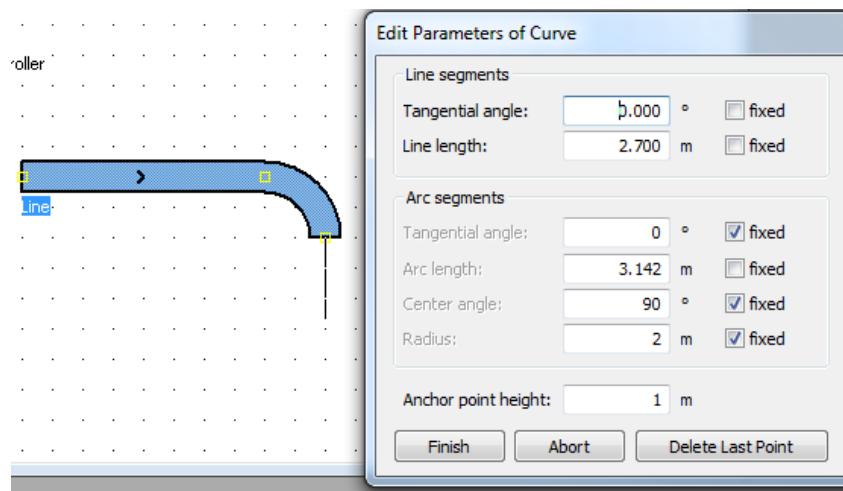


Fig. 6.8 Arc segments

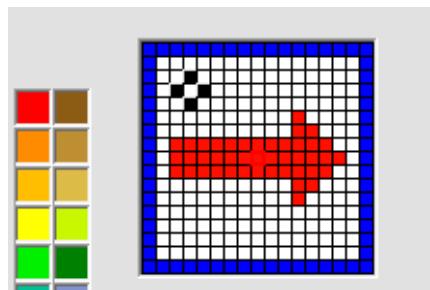
## 6.2 AngularConverter and Turntable

The AngularConverter and the turntable assist you in modeling curves or junctions on lines. Plant Simulation offers three options:

1. Append a corner point to the line and extend the line in a 90-degree angle. In the absence of SimTalk, you cannot implement a special (higher) time for the transfer.
2. You can use the object AngularConverter. This will help you to simulate processes in which the part is transported to a certain point, stops and then accelerates again at an angle of 90 degrees. The retardation and acceleration times will be considered as time (e.g. four seconds). The entity will not be rotated during this process.
3. If the part is to be rotated during transfer (e.g. via a robot or a turntable), you can use the object turntable.

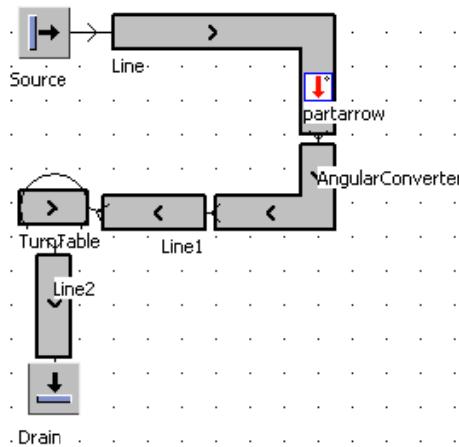
### Example: Turntable and AngularConverter

Here is a small example comparing the various solutions: Duplicate an entity, and rename it “partarrow”. Change the icon so that it matches the picture in Fig. 6.9.



**Fig. 6.9** Partarrow icon

Create a frame according to Fig. 6.10.



**Fig. 6.10** Example frame

Enter the following settings: The source generates the MU partarrow at intervals of one minute. Remain the basic settings for all other objects. Start the simulation and track the movements of the part.

### 6.2.1 Settings of the AngularConverter

You can select different lengths and the associated speeds (Fig. 6.11).

Entry length:	2	<input type="button" value="m"/>
Exit length:	3	<input type="button" value="m"/>
Entry speed:	1	<input type="button" value="m/s"/>
Exit speed:	1	<input type="button" value="m/s"/>

Fig. 6.11 Attributes of the AngularConverter

The moving time (Tab Times) is the time that the converter needs to switch from one direction to the other (Fig. 6.12).

Moving time:	Const	0:03	<input type="button" value=""/>
--------------	-------	------	---------------------------------

Fig. 6.12 Moving time

### 6.2.2 Settings of the Turntable

The turntable accepts one part, rotates it by 90 degrees and moves it in the direction of the connector. If you select "Go to default position" (and possibly enter an angle), the turntable returns to this position after moving the part. If the option is not selected, the turntable rotates only when the next part is ready to be moved (Fig. 6.13).

Attributes	Times	Failures	Controls	Exit Strategy	Statistics	Curve	User-def
Length:	2	<input type="button" value="m"/>	<input type="button" value="Entry Angle Table"/>	<input type="button" value="Exit Angle Table"/>			
Rotation point:	1	<input type="button" value="m"/>					
Conveyor speed:	1	<input type="button" value="m/s"/>					
Rotation time per 90°:	0:04	<input type="button" value=""/>					
Rotate when:	Centered	<input type="button" value=""/>					
<input checked="" type="checkbox"/> Go to default position	<input type="button" value=""/>	Default angle:	0	<input type="button" value=""/>	<input type="button" value=""/>		

Fig. 6.13 Settings for the turntable

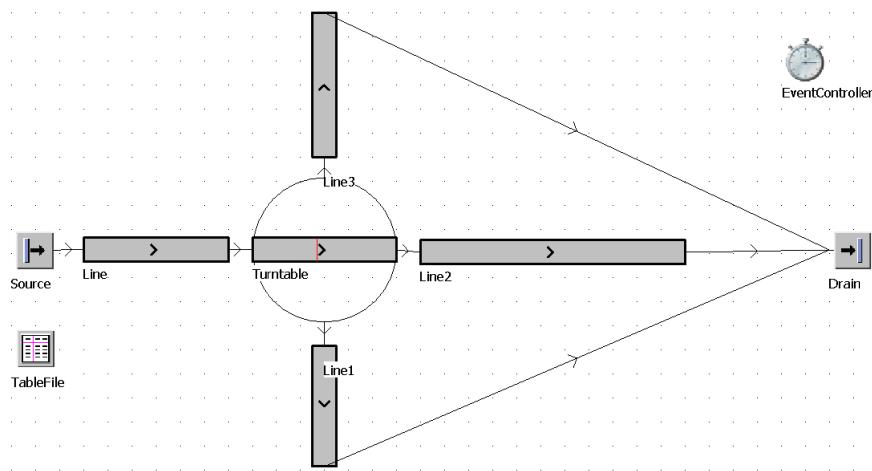
### 6.2.3 Turntable, Select User-defined Exit

Using a sensor control located on the turntable, you can select an exit. To model realistic behavior, you must proceed step by step:

1. Stop the line of the turntable
2. Select an exit (setDestination)
3. Wait until the rotation is complete
4. After the rotation, start the line again

#### Example: Turntable: User-defined exit

In the example, use a turntable to distribute MUs by name to the successors. Create a frame as in Fig. 6.14



**Fig. 6.14** Example Frame

The source produces MUs at intervals of 25 seconds. Set the source to MU selection: Sequence cyclical. Assign TableFile as Table. Ensure the settings in TabFile according to Fig. 6.15.

	object 1	integer 2	string 3	ta 4
string	MU	Number	Name	At
1	.MUs.Entity	1	P1	
2	.MUs.Entity	1	P2	
3	.MUs.Entity	1	P3	

**Fig. 6.15** Settings

The turntable is intended to distribute the MUs as follows: P1 to Line1, P2 to Line2, P3 to Line3. Create one sensor on the turntable and create a method using the key F4. The sensor control should have the following content:

```

is
  temp_speed:speed;
do
  temp_speed:=self.~.speed;
  --stop line
  self.~.speed:=0;
  --select destination and turn
  if @.name="P1" then
    self.~.setDestination(Line1);
  elseif @.name="P2" then
    self.~.setDestination(Line2);
  else
    self.~.setDestination(Line3);
  end;
  --wait for completing the turning
  waituntil self.~.resWorking=false prio 1;
  --start line
  self.~.speed:=temp_speed;
end;

```

## 6.3 The Turnplate

Version 10 of Plant Simulation provides the Turnplate.

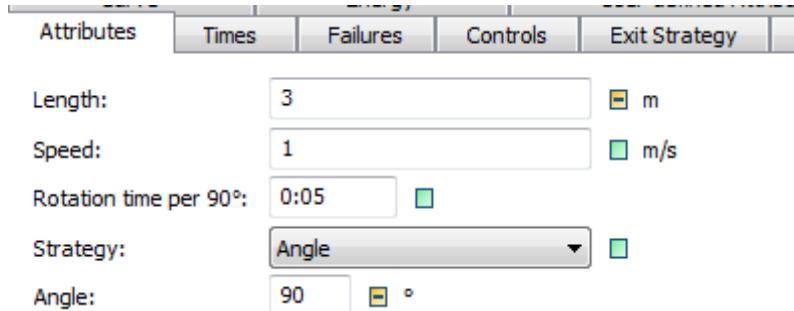
### 6.3.1 Basic Behavior of the Turnplate

The Turnplate transports the MU until its booking point is located above the center of the Turnplate. The Turnplate then rotates the MU for a number of degrees in a set time. Thereafter, the parts are transferred with a corresponding conveying direction to the next block. The difference to the turntable is that the part is also rotated at the "straight running". A reverse rotation of the Turnplate is not necessary. The use of this block is of interest where parts in conveyor technology must be turned for a subsequent processing to change the conveying direction. Plant Simulation rotates in full 90-degree steps.

### 6.3.2 Settings of the Turnplate

If you are using a turnplate, you must create the diameter and the direction of the turnplate similar to the width of the MU and the direction of the line. Click the icon for the turnplate in the toolbox, and then click within the network on the point where the outer edge of the turnplate should be located. The mouse icon will

change and the dialog "Edit curve parameters" appears. Click on the position of the right edge. Plant Simulation then generates the turnplate with the appropriate diameter. If you want to use a custom icon, deactivate the curve in the tab Curve (unchecked the checkbox Active). The most important settings of the Turnplate you will find in the tab Attributes (Fig. 6.16).



**Fig. 6.16** Attributes of the Turnplate

**Length:** Enter here the diameter of the turnplate. By default, the curve mode and the setting "transfer length" are active. If you change the diameter, then the size of the symbol in the network changes as well. If the length of the turnplate is insufficient for rotating the MU, an error message is displayed. The necessary length of the rotating plate is dependent on the length and booking point of the MUs. If your MU e.g. is two meters long and you have set the booking point (length) at 0.4 meters, then the turnplate needs a diameter (length) of 3.2 meters to turn the MU ( $\text{abs}(\text{length}-\text{booking point}) * 2$ ). To better observe this behavior, you should set the reference point of the MUs analogous to the booking point.

**Speed:** At the speed you set here, the part is transported to the center of the Turnplate, and after the rotation further until the end.

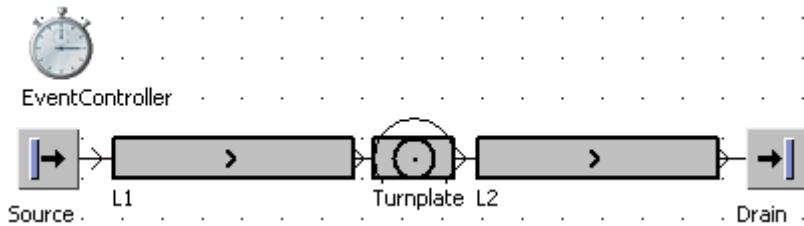
**Rotation Time Per 90 Degrees:** The Turnplate only rotates a full 90 degrees. Different values are rounded up. The time for the rotation is then calculated from the number of integer multiples of 90 degrees with the entered time per 90 degrees. You can also enter values greater than 360 degrees in order to simulate multiple rotations.

**Strategy:** The strategy determines how the angle of rotation is determined. Angle means that the value in the field Angle is determinative. You can define, using the strategies MU-attribute and MU-name, lists with associated angles or calculate the angle using the strategy method.

### Example: Turnplate

In the following example, we will determine which total cycle time is caused by the Turnplate. For this, the following experimental setup is necessary: In a production, bars are processed. The bars have a length of 80 cm and a width (diameter) of 40 cm.

Between two machining operations, the orientation must be changed (from longitudinal to transverse). Create a frame as in Fig. 6.17.



**Fig. 6.17** Example frame

The source generates one part every seven seconds. The conveyor lines are each six meters long. The conveyor line L1 has a speed of 1 m/s and the conveying line L2 of 0.25 m/s. Modify the attributes of the entity in the class library as follows: length: 0.8 meters, width 0.4 meters, booking point width: 0.2 meters; booking point length: 0.4 meters. The turnplate has a diameter of two meters, a speed of 1 m/s and needs five seconds for a rotation of 90 degrees. The Angle is 90 degrees. Shortly after you start the simulation, a jam occurs. So, the turnplate seems to need more than seven seconds (2 meters \* 1 m/s + 5 seconds rotation). On closer inspection, the rotating plate requires the following times:

Description	Time
Enter until the booking point is in the middle of the plate (one meter to the middle + additional 0.4 meters to the booking point)	
Rotation	5 seconds
Movement of the booking point to the entrance of the next conveyor section (one meter)	
Until the departure of the part (the part must completely leave the turnplate), the part moves at the speed of the next conveyor section (0.2 meters / 0.25 m/s, 0.8 seconds).	

The total cycle time for the Turnplate is 8.2 seconds. If you set the interval of the source to 8.2 seconds, no jam takes place.

## 6.4 The Converter

The converter helps you to model conveyor systems in the z-axis (e.g. lift). The converter is available from Version 10. It carries an MU vertically up or down and then transfers the MU in the same or a different conveying direction to the successor. Using this object, you can model devices for lifting and lowering within the conveyor system. In the case that the conveying direction is changed, the

converter also simulates the behavior that the converter has to first return to its initial position before the next part can be transported.

### Settings of the Converter

The converter is a length-oriented block. The best way to generate the converter is to do this similar to the line. Click in the toolbar the icon of the converter and then click in the network on the position of the beginning of the converter. Create the converter through another click on the frame at the correct length (end position) in the conveying direction. The conveying direction is displayed on the icon of the converter as an arrow. Next, connect the converter in the direction of the material flow. The following settings are possible in the tab Attributes (Fig. 6.18).

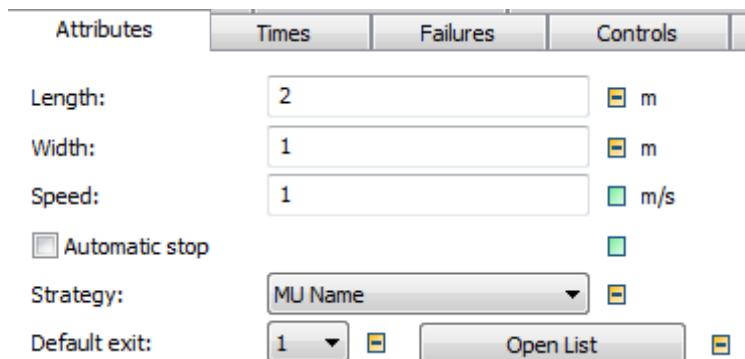


Fig. 6.18 Settings of the converter

Another way of setting the length is by dragging the markers of the converter if the "transfer length" option in the tab curve is activated. Using strategy, you determine the direction of the movement of the MU. The choices are: straight (further in the conveying direction), MU name, MU attribute (table with assignments of the successor to the names or attributes) or method (the successor is returned by a method). The successors are numbered as in Fig. 6.19.

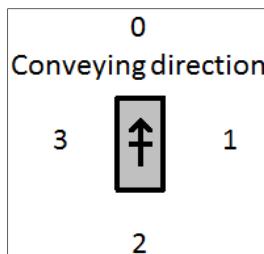


Fig. 6.19 Successor converter

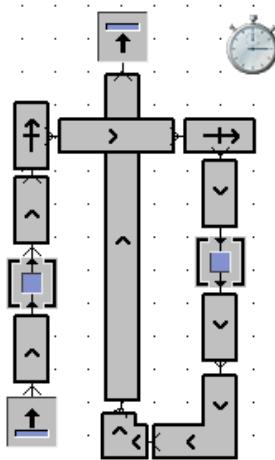
The cycle time on the converter is calculated analogous to the turnplate (default setting: entity: 0.8 x 0.8; the booking point length and width: each 0.4 meters; length of the converter: two meters; width: one meter; moving time: three seconds):

1. Transport of the MU until the booking point is above the center (1 meter + 0.4 meters at 1 m/s) 1.4 seconds
2. Moving time: three seconds
3. Transport transversely: booking point to the edge 0.5 seconds, until exiting (the remaining part, booking point to the edge) 0.4 seconds

In sum, the converter needs with default settings of 5.3 seconds for one transfer operation.

### Example: Converter

In a production, the production flow runs with the help of two lifts through different levels in the plant. Create a frame as in Fig. 6.20.



**Fig. 6.20** Example frame

All blocks are connected in the direction of the material flow by connectors. The source produces parts at an interval of one minute. All other blocks remain in their default settings. Proceed when inserting the converter as follows: Click on the converter icon in the Toolbox. Then click in the frame at the start point of the converter. The mouse icon changes to indicate the curve mode. Converters are always drawn in the direction of material flow—i.e. the converter left must be drawn from the bottom up. Click on the end point of the converter (two meters). The exact length can be adjusted later in the converter dialog. Connect the right side of the first converter to the next conveyor line, etc. You must define Exit 1 as the exit in the converter dialog (0 is straight; 1 is offset by 90 degrees clockwise). A setting as in Fig. 6.21 is sufficient for this.

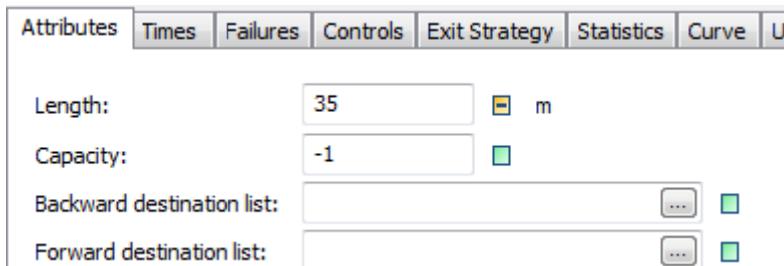


**Fig. 6.21** Default exit

If you set only one exit, then it is sufficient to define a default exit. Entries in the list are necessary if you want to transfer MUs to different exits (directions). Highlight one side of the conveyed MU icon to observe the direction of the transport.

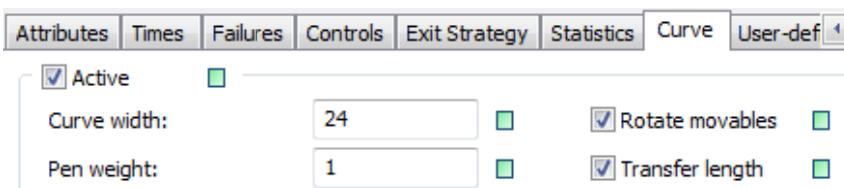
## 6.5 The Track

The track is a passive object for modeling transport routes. The transporter is the only moving object that can use the track. The dwell time on the track is calculated using the length of the track and the speed of the transporter. MUs cannot pass each other on the track (retain their entrance order—FIFO). If several transporters with different speeds are driving on the track (a faster one catches up with a slower one), a collision occurs. The faster transporter automatically adjusts its speed to that of the slower one. With a capacity of -1 (Fig. 6.22), the maximum capacity of the track is determined by the length of the track and the length of the transporters (length of ten meters and one meter per transporter results in a maximum of ten transporters); otherwise, the capacity is limited by what you enter.



**Fig. 6.22** Attributes of the track

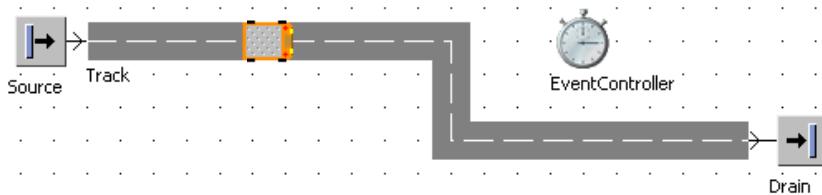
**Backward/Forward destination list:** A track can connect several workstations. You can specify which stations have to be covered on the route (forward and backward). Plant Simulation determines the length of the track if you activated the checkboxes Active and Transfer Length on the tab Curve (Fig. 6.23).



**Fig. 6.23** Tab curve

### Example: Track

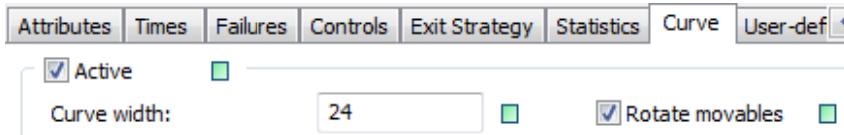
Create a frame as in Fig. 6.24.



**Fig. 6.24** Example frame

You have to "convince" the source to produce transporters. Open the source by double-clicking it. Enter a time of one minute as the interval for the production. Select .MUs.Transporter as the MU. When you start the simulation, the source produces a new transporter every minute and moves it onto the track. The transporter moves with the speed you entered on the track and will be moved to the drain at the end of the track. The drain destroys the transporter.

Clear the option "Rotate movables" on the tab Curve (Fig. 6.25) and test what happens.



**Fig. 6.25** Rotate movables

The option of "Rotate movables" animates the transporter, so that it always points forward (the front always points in the direction of movement). Therefore, the icon of the transporter rotates. Try it once with a curve (as with the line: context menu, Append points and insert the curve using **CTRL + SHIFT + mouse click** into the frame).

## 6.6 Sensors on Length-Oriented Blocks

You can equip length-oriented blocks with sensors, where you can trigger methods.

### 6.6.1 Function and Use of Sensors

Lines and tracks can be very long. Therefore, it may be useful to trigger methods if the MU is located some distance before the end or to set a few breakpoints

regarding which methods should be executed. For this purpose, you can define sensors. The sensors act like switches. When an MU crosses (forward or backward) the sensor, the switch is activated and a method is called. In the method (control), you have to determine what should happen at this position. A small example illustrates this.

### Example: Sensors, color sorting

Within a production facility, parts bearing different colors arrive. With the help of cameras and sorting facilities, the parts will be distributed to different color-homogenous lines. We do not want to simulate the sorting facility. Create a frame as in Fig. 6.26. Perform the following settings: main\_line: 35 meters; L\_red, L\_blue and so on: five meters. All lines are accumulating with a speed of 1 m/s and no acceleration. All L-lines are connected to drains. The drains have a processing time of zero seconds. The source is connected to the main\_line.

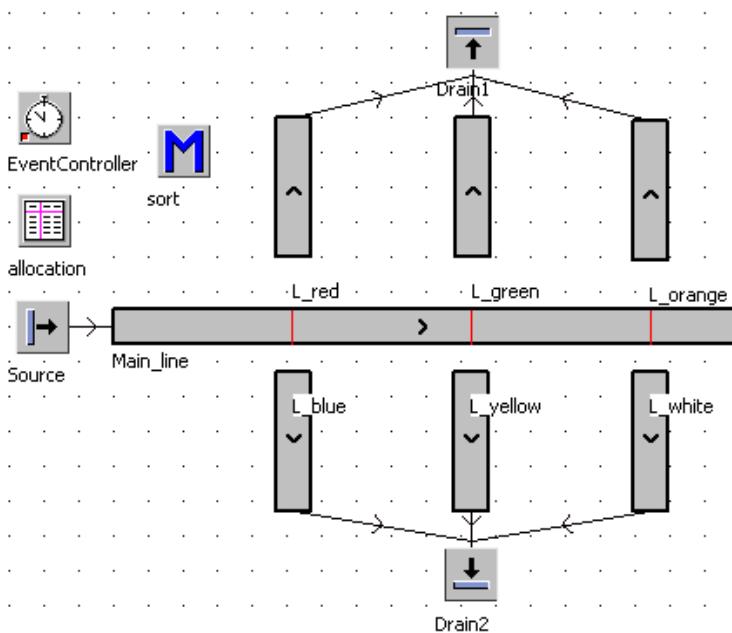


Fig. 6.26 Example frame

Duplicate six entities in the class library. Rename them as red, blue, green, yellow, orange and white. Type in an icon size of 11 x 11 pixels and color the icons with the respective colors. The source should randomly produce these parts

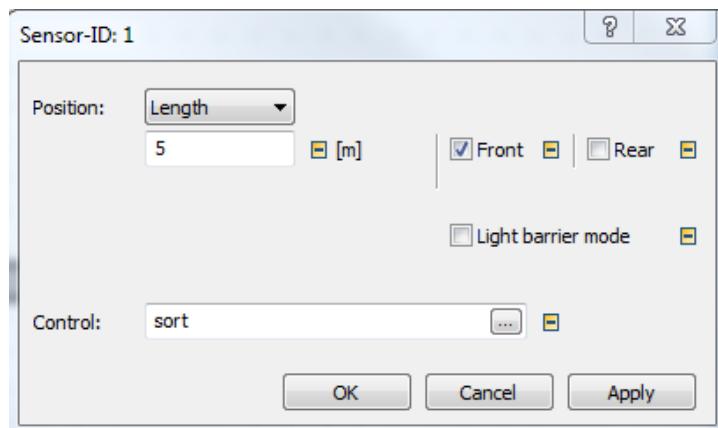
with a percentage of 16.7 per cent each. Select the following settings in the source: interval: 0.5 seconds; MU Selection: random; table: allocation. Enter the data from Fig. 6.27 into the table allocation.

	object 1	real 2
string	MU	Frequencies
1	.MUs.red	0.17
2	.MUs.green	0.17
3	.MUs.blue	0.17
4	.MUs.yellow	0.17
5	.MUs.white	0.17
6	.MUs.orange	0.17

**Fig.6. 27** Table allocation

Note: Drag the MUs from the class library to the table to enter the absolute paths of your MUs. Depending on the location of the MUs in the class library you may have other addresses in the table.

Insert three sensors on the main\_Line (10 m, 20 m and 30 m each rear). Assign the method sort to all sensors. Proceed as follows: Click the button Sensors on tab control in the dialog sensor of the main\_Line. Then, click the button New and enter a position (e.g. ten meters). Decide whether the front or the rear (checkbox) of the MU should trigger the sensor. Select the control “sort” (Fig. 6.28).



**Fig. 6.28** Sensor New

Complete the sensor list as in Fig. 6.29.

ID	Position	Front	Rear	L.	Path
1	5m	x			sort
2	10m	x			sort
3	15m	x			sort

**Fig. 6.29** Sensor list

The method "sort" will check the name of the MU. If the MU "red" arrives at Sensor 1, then the MU is to be moved to the line L\_red, and so on.

```
(sensorID : integer)
is
do
  if sensorID=1 then -- first sensor
    -- red to L_red, blue zu L_blue
    if @.name="red" then
      @.move(L_red);
    elseif @.name="blue" then
      @.move(L_blue);
    end;
  elseif sensorID=2 then
    if @.name="green" then
      @.move(L_green);
    elseif @.name="yellow" then
      @.move(L_yellow);
    end;
  elseif sensorID=3 then
    if @.name="orange" then
      @.move(L_orange);
    elseif @.name="white" then
      @.move(L_white);
    end;
  end;
end;
```

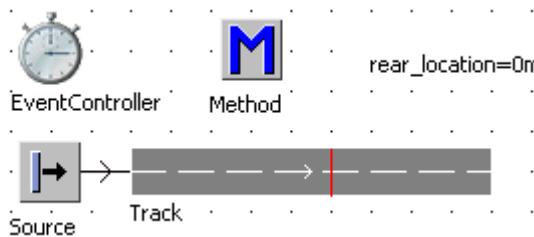
The parts should now be color-sorted on the L\_-lines.

## 6.6.2 Light Barrier Mode

Starting with Version 10, there is a light barrier mode when simulating sensors on tracks and conveyor lines. If you select the option "light barrier mode", the sensor will be triggered while moving forward with the front and at the backward movement with the rear.

### Example: Light barrier mode

Create a frame as in Fig. 6.30.

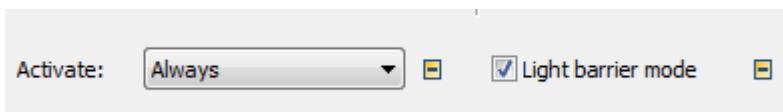


**Fig. 6.30** Example frame

The source generates a transporter once. Create on the track a sensor and assign to the sensor the method as control. The method determines the position of the rear end and stores this in the variable rear\_location.

```
(SensorID : integer; Front : boolean)
is
do
  rear_location:=@.rearPos;
end;
```

Let the car drive forward and backward over the sensor (double-click on the transporter and set or remove the checkmark next to "Backwards"). If you have not selected the "light barrier mode" option, then the rear position when triggering the method is independent of the direction of the transporter. Now, select the "light barrier mode" option (Fig. 6.31).



**Fig. 6.31** Light barrier mode

If you now test the behavior, then the rear positions differ depending on the direction of the vehicle (choose front for correct operation).

**Note:** If you want to map the behavior of a light barrier correctly, then it may be helpful not to use the light barrier mode. Instead, select both front and rear. A real light barrier is triggered twice: the first time if the front or rear of the transporter interrupts the laser beam (value from true to false), the second time when the transporter leaves the position and the laser beam falls on the sensor again (value from false to true).

### 6.6.3 Create Sensors Automatically

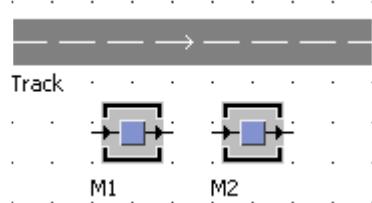
You can also create sensors using SimTalk. For this, you need the position of the sensor on the track or the line and the method to be triggered. Using the segment tables of the track and the line, you can determine the positions of these elements in the frame. With this information, you can develop a method to generate the sensors automatically by dragging objects on the length-oriented elements. The following methods are necessary (Table 6.1):

**Table 6.1** Method for generating sensors

Method	Description
<path>.createSensor(<real>,<string>,<object1>);	Generates a sensor and returns the ID of the sensor; you must pass the position (real), a string for the position type ("length", "relative") and a reference to the method to be executed
<path>.sensorID(<integer>).position	Sets and reads the position of the sensor with the ID <integer>
<path>.scalingFactor	Returns the scaling factor of the frame (e.g. size of a pixel in meters)
<path>.segmentsTable	Provides access to the segment table of the length-oriented element (see tab Curve button Segments)
Str_to_obj(<string>)	Creates an object reference from a string (e.g. object path)
<path>.xPos;	Returns the x coordinate of the object within the frame; the base is the upper left corner of the frame

#### Example: Create sensors using SimTalk

You should create a function that will generate sensors on a track through drag and drop. The sensor number and a reference to the object should be stored in an internal table of the track. If the object was already been entered in the table, then the entry in the list will be deleted and a new entry will be created. Duplicate a track in the class library. Using the duplicated track, create a frame according to Fig. 6.32.

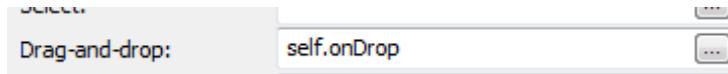


**Fig. 6.32** Example frame

Create three user-defined attributes in the duplicated track in the class library:

- onDrop (method)
- sensorControl (method)
- sensorList (table)

Format the first column in the table as an integer and the second column as an object. Select in the track Tools – Select Controls – Drag-and-Drop and assign onDrop to the control (Fig. 6.33).



**Fig. 6.33** Drag and drop control

To simplify the example, the programming is shown for horizontally running tracks only. An extension for vertical tracks is possible without major problems. The method onDrop must fulfill the following tasks: If the object is already entered in the sensor list, the sensor with the corresponding ID will be moved to a new location; otherwise, a new sensor will be created and a corresponding entry in the table will be generated. The sensor position is calculated from the x-position of the object (taking into account the scaling factor). So, the method onDrop might resemble the following:

```
(obj:string)
is
  posi:real;
  station:object;
  found:boolean;
  sensorID:integer;
do
  --create reference
  station:=str_to_obj(obj);
  -- calculate x position
  posi:=(station.xPos-self.~.segmentsTable[1,1])*
  self.~.location.scalingFactor;
  -- look in the sensorList
  self.~.sensorlist.setCursor(1,1);
  found:=self.~.sensorlist.find(station);
  -- found, move sensor
  if found then
    self.~.sensorID(self.~.sensorList[1,
    self.~.sensorlist.zeigerY]).position:=posi;
  else
    -- create new Sensor
    sensorId:=
    self.~.createSensor(posi,"Length",
    "sensorControl");
```

```
-- create entry
self.~.sensorlist.writeRow(1,
    self.~.sensorList.yDim+
    1,sensorID,station);
end;
end;
```

## 6.7 The Transporter

With the help of the transporter, you can model many transport facilities. The transporter moves on the track with a set speed forward or in reverse. Using the length of the track and the speed of the transporter, the time that the transporter spends on the track is calculated. At the exit, the track transfers the transporter to a successor. Transporters cannot pass each other on a track. If a faster transporter moves up close to a slower one, then it automatically adjusts its speed to that of the slower transporter. When the obstacle is no longer located in front of the transporter, the transporter accelerates to its previous speed.

### 6.7.1 Attributes of the Transporter

Create an object forklift (duplicate a transporter, speed 1 m/s) in the class library. Open the object by double-clicking it. Fig. 6.34 Attributes of the shows the attribute dialog from Plant Simulation Version 11.

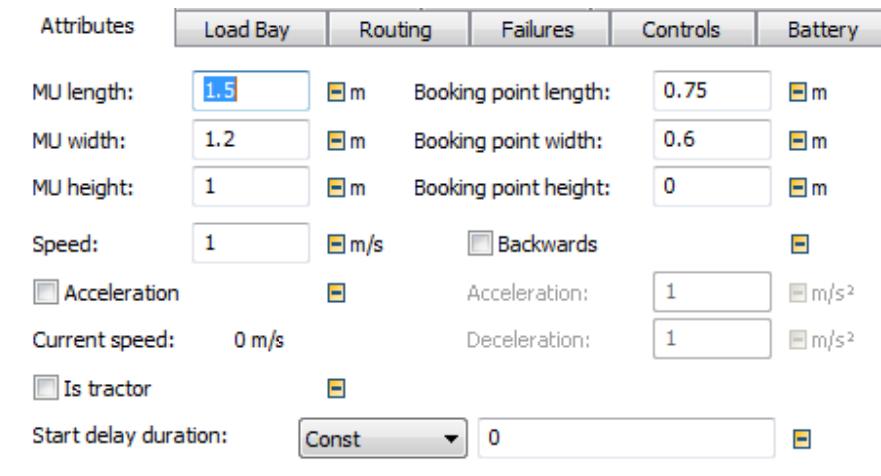


Fig. 6.34 Attributes of the transporter

**MU Length/Width:** The length of the transporter must be smaller than the length of the track, if you want to create a transporter on a track. The capacity of the

tracks (setting capacity = -1) is calculated as the length of the tracks divided by the length of the transporter.

**Booking Point Length:** The booking points determine the position from which the transporter is booked to the next block. This is important if you e.g. want to access the transporter using `<path>.cont`. If a vehicle is moved to the successor (by standard), only the front of the transporter is moved to the next object. The transporter then moves at the set speed on the new track, until it is located across the entire length on the next track. At the point where you can access the transporter on the track, this determines the booking points (length and width depend on the conveying direction).

**Speed:** Enter the speed with which the transporter moves on the object track. The speed is a positive value (data-type real). If you set the speed to zero, the transporter stops. You can also simulate acceleration and deceleration of the transporter (option Acceleration).

**Backwards:** This option activates the movement of the transporter in reverse on the track (it also can be called by a method e.g. to drive back the transporter after unloading).

Starting with Version 10, the transporter has three different types of load bays (Fig. 6.35).

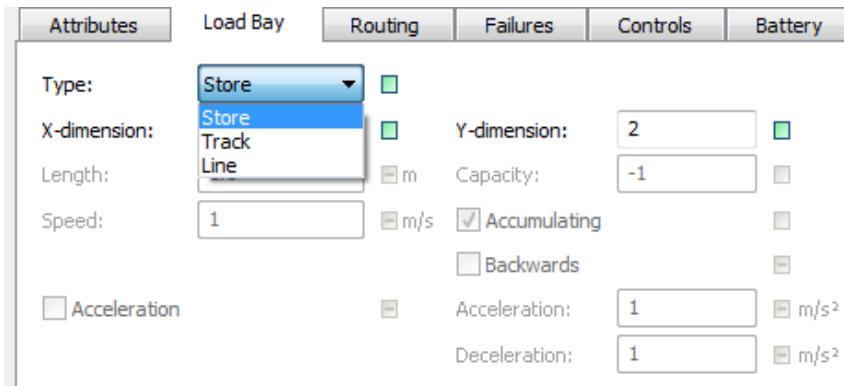


Fig. 6.35 Transporter load bay types

**Type Store:** If a type store is selected, the position of MUs on the load bay is determined by xy coordinates.

**Type Track:** The vehicle then has a length-oriented structure. You can then, for instance, drive with another transporter on this transporter. The capacity of the load bay is normally determined by the length of the load bay and the length of the transporters that you let drive onto the load bay of the transporter. Of course, you can also limit the capacity of the load bay (by setting a value greater than zero in

the field capacity). Analogous to the track, you have the option to place sensors on the length-oriented load bay of the transporter and assign methods to the sensors.

**Type Line:** Here, the load bay actively transports MUs (forward and backward). This is particularly useful when you simulate transporters that are using e.g. roller conveyors on their load bay for loading or unloading.

**Load Bay Length:** This refers to the length of the length-based load bay (you can attach sensors and use the length-based load bay as a track for e.g. plate cars, car transporters, loading portals, cranes, etc.)

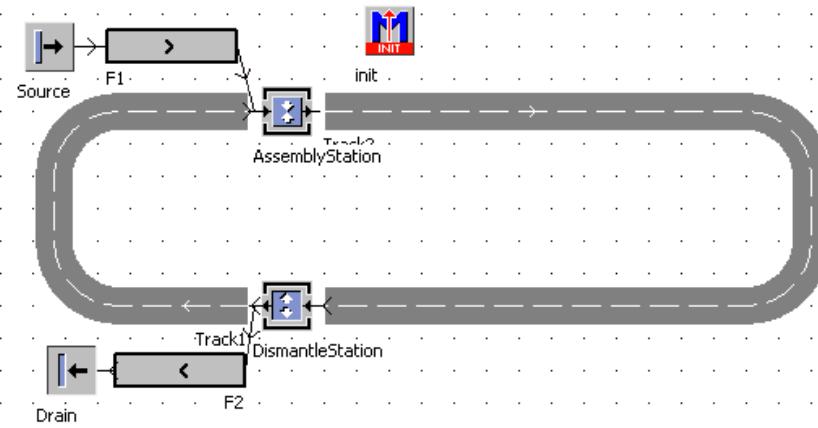
**Capacity:** Here, you enter the number of MUs that can fully or partially be located on the transporter. -1 means no limit; the length of the individual MUs determines the capacity.

### 6.7.2 Load and Unload the Transporter Using the AssemblyStation and the DismantlingStation

You can use the AssemblyStation in order to load the transporter and the DismantlingStation to unload the transporter again. The transporter and charged MUs have to access the assembly station via different predecessors. The transporter is the main part in this process and must have enough capacity to load the necessary number of MUs. In the following example, an entity is to be loaded on a transporter and unloaded from the transporter again.

#### Example: Loading and unloading the transporter 1

Create a frame like in Fig. 6.36.



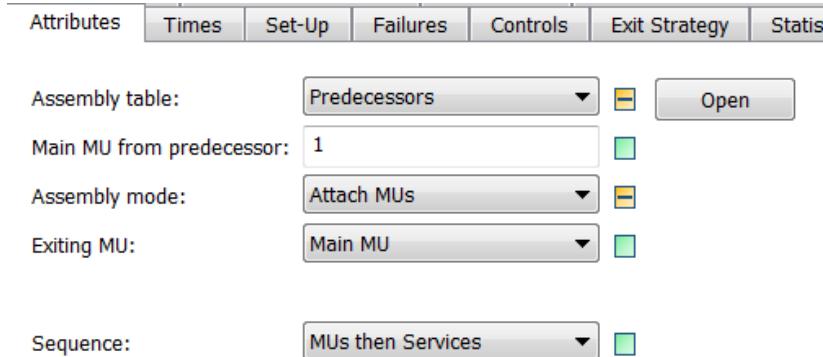
**Fig. 6.36** Example frame

First, connect track1 with the assembly station and then F1. Analogously connect first track1 with the disassembly station, then F2. The transporter will be

inserted in the frame using the init method. To do this, the init method should have the following content:

```
is
do
    .MUs.transporter.create(track1);
end;
```

The assembly station has to work in the mode "Attach MUs". Configure the assembly station according to Fig. 6.37.



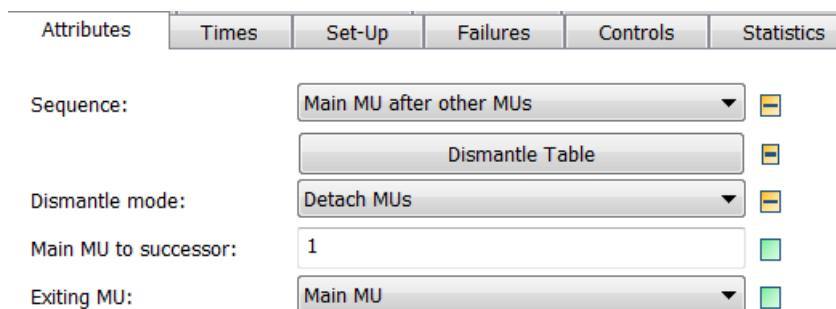
**Fig. 6.37** Settings of the AssemblyStation

The type (origin) and the number MUs for loading can be determined in the assembly table. In our case—the loading of one MU from Predecessor 2—the assembly table should resemble that shown in Fig. 6.38.

	Predecessor	Number
1	2	1

**Fig. 6.38** Assembly table

For the AssemblyStation, you can also set the time that is caused by the loading process. The DismantleStation has to work in the mode "Detach MUs". Ensure the settings in the DismantleStation as shown in Fig. 6.39.



**Fig. 6.39** Settings of the DismantleStation

The dismantle table contains target specifications for individual MU classes. To transport the unloaded MU to Successor 2, you must make the adjustment from Fig. 6.40.

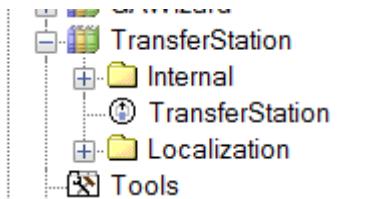
	MU	Number	Successor
1	.MUs.Entity	1	2

**Fig. 6.40** Dismantle table

The necessary time for unloading can be set in the in the tab Times.

### 6.7.3 *Load and Unload the Transporter Using the TransferStation*

The TransferStation is a user object and is located in the class library in the Tools folder (Fig. 6.41).



**Fig. 6.41** Class Library

You can use the TransferStation to load and unload transporters.

### Example: TransferStation

Create a frame as in Fig. 6.42.

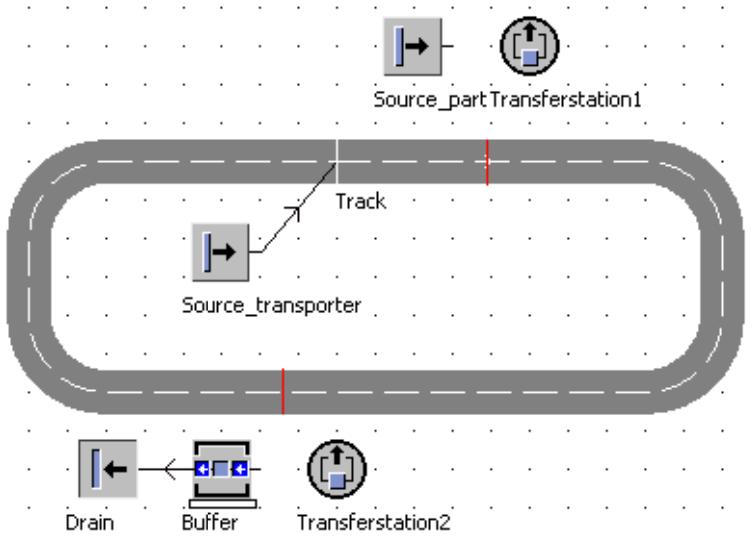


Fig. 6.42 Example frame

Connect the end and the beginning of the track so that the transporter can drive "in a circle". Set the source `source_transporter` so that it produces just one transporter. The source `source_part` creates entities at an interval of one minute. Adjust the capacity of the transporter in the class library to one MU. Add sensors at the loading and unloading position. You do not need to assign controls to these sensors. In the TransferStations, set the source and the destination of the transfer and set the time that is necessary for this. For example, for the loading, the source of the transfer process is `Source_part` and the destination is the track—more specifically, the sensor position of the loading position. The necessary adjustments for loading the transporter are presented in Fig. 6.43.

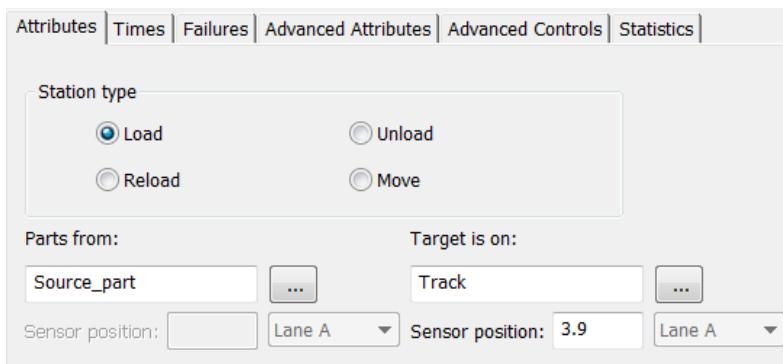


Fig. 6.43 Settings of the Transferstation loading

For unloading the transporter, enter the track as the origin of the transfer and the buffer as the destination (see Fig. 6.44).

The time can be set in the tab Times. With the TransferStation, you can also model load operations between two transporters (mode reload) or the moving of the transporter itself from one track to another (mode move).

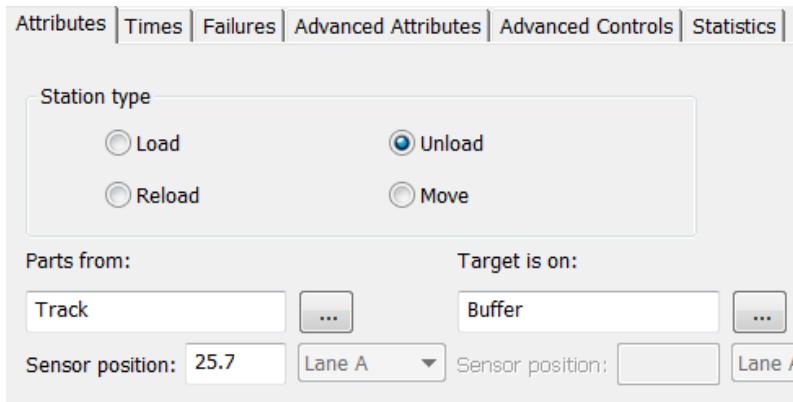


Fig. 6.44 Transferstation unloading

#### 6.7.4 Load and Unload Transporter Using SimTalk

The loading and unloading of the transporter works analogous to the container, triggered e.g. by an exit control or a sensor control on the track.

##### Example: Unload transporter

You should unload the content of a transporter to machine M2.

```
is
do
  @.cont.move(M2);
end;
```

Explanation: In this case, @ means the transporter. You get access to the content of the transporter using the method cont (@.cont). The method cont returns a pointer to the MU (or void, if the transporter is empty). The MU can be moved to the machine M2 using the method move (...move (M2)).

#### 6.7.5 SimTalk Methods and Attributes of the Transporter

In the Table 6.2, you will find a selection of important SimTalk methods and attributes of the transporter.

The transporter also provides a number of methods and properties that deal with the battery operation and related problems.

**Table 6.2** Transporter methods and attributes

Method/Attribute	Description
<path>.create(<destination>)	Method creates a new object; on length-oriented blocks, you can determine the initial position of the object
<path>.create(<destination>, <length>)	
<path>.startPause;	Immediately pauses the transporter and sets the attribute pause to the value true; when a parameter is passed (integer greater than zero), it determines after which time (in seconds), the transporter changed back to the non-paused state
<path>.startPauseIn(<time>)	Pauses the transporter after the period defined in <time> has passed
<path>.collided	Collided returns true if the transporter is collided with another transporter
<path>.xDim	Sets/gets the dimension of the matrix load bay
<path>.yDim	
<path>.backwards	Determines whether the vehicle travels backward (true) or forward (false)
<path>.speed	Specifies the speed with which the transporter moves on the track; the speed must be equal to or greater than zero—if you set the speed to zero, then the transporter stops
<path>.destination	Sets/gets the destination of the transporter
<path>.stopped:=true/false	Starts and stops the transporter
<path>.movingSpeed	Returns the actual speed of the transporter; this attribute is observable

### 6.7.6 Stopping and Continuing

To stop and continue after a certain time is the normal behavior of the transporter. While the transporter waits, you can e.g. load and unload the transporter or recharge its battery. In SimTalk, you use the attribute `Stopped` to stop the transporter and make it continue on its way.

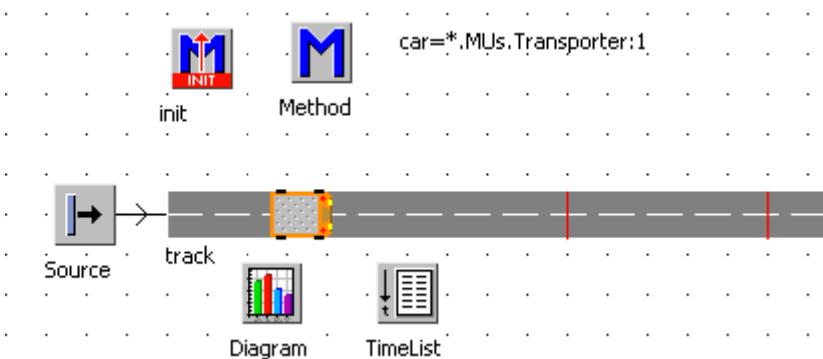
Syntax:

```
@.stopped:=true; --stop the transporter
@.stopped:=false; --the transporter drives again
```

Another possibility to stop the vehicle is to set the speed at zero. The vehicle then slows down with the set acceleration and stops. You can start the vehicle again by setting the speed at its original value. In this way, you can take into account acceleration and slowdown in the simulation. To demonstrate these two options, you can use the following two examples.

### Example: Stopping the transporter

Create a frame like in Fig. 6.45.



**Fig. 6.45** Example frame

Set the values in the transporter in the class library according to Fig. 6.46.

Attributes	Load Bay	Routing	Failures	Controls	Battery
MU length:	1.5	<input type="button" value="m"/>	Booking point length:	0.75	<input type="button" value="m"/>
MU width:	1.2	<input type="button" value="m"/>	Booking point width:	0.6	<input type="button" value="m"/>
MU height:	1	<input type="button" value="m"/>	Booking point height:	0	<input type="button" value="m"/>
Speed:	1	<input type="button" value="m/s"/>	<input type="checkbox"/> Backwards		<input type="button" value=""/>
<input checked="" type="checkbox"/> Acceleration		<input type="button" value=""/>	Acceleration:	0.5	<input type="button" value="m/s&lt;sup&gt;2&lt;/sup&gt;"/>
Current speed:	0 m/s		Deceleration:	0.5	<input type="button" value="m/s&lt;sup&gt;2&lt;/sup&gt;"/>

**Fig. 6.46** Attributes of the transporter

The source should create one transporter. This works with the adjustment as in Fig. 6.47.

The transporter has to stop after ten, 15 and 20 meters and start driving again after five seconds. Insert sensors on the track at these positions and assign to the sensors the method as control. To study the behavior, we need a speed-time chart. We can create this using a TimeSequence in which we record the values of the attribute movingSpeed at regular intervals. Format the second column of the TimeSequence as data-type speed. Make the settings in the TimeSequence according to Fig. 6.48.

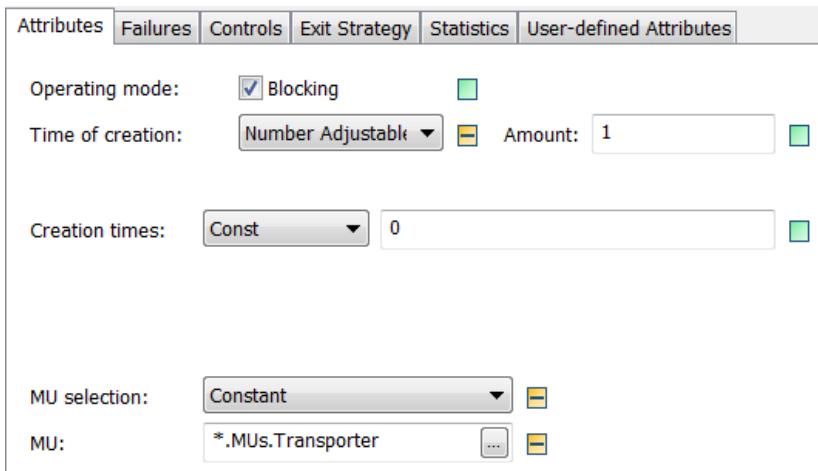


Fig. 6.47 Setting source

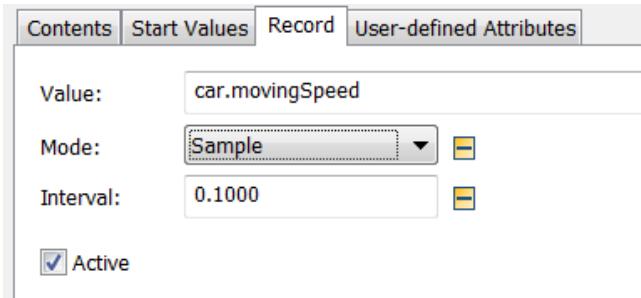


Fig. 6.48 Settings of TimeList

Recording the speed of the transporter with a TimeSequence causes a problem. The first value will be recorded at time 0. At this time, no transporter exists in the model. We can disable the TimeSequence and activate it again after the creation of the transporter in order to avoid this problem at the beginning of the simulation (init). Insert into the method init the following content:

```
is
do
  --delete the content
  timeList.delete;
  -- deactivate the TimeSequence
  timeList.active:=false;
end;
```

After the generation of the transporter we need to set the reference to the transporter and activate the TimeList. To do this, insert an exitcontrol (rear) into

the source (press key F4 in the field ExitControl). The exitcontrol should have the following content:

```
is
do
  car:=@;
  TimeList.active:=true;
end;
```

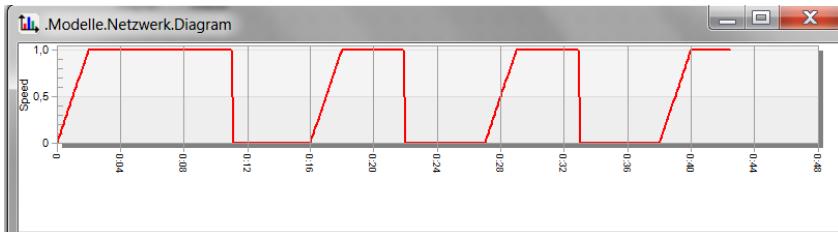
To display the TimeList in the chart, drag the TimeList on the chart. Select in the tab Display: category: XY graph; chart type: line; data: in column.

*Variant 1: stop with stopped: = true*

Add sensors at the stop positions and start at that position a method which stops the vehicle (and starts again after five seconds). The deceleration is not considered. If you start the transporter and acceleration is activated, the transporter accelerates to the set speed. The method (sensor control) could appear as follows:

```
(SensorID : integer; isFront : boolean)
is
do
  @.stopped:=true;
  wait(5);
  @.stopped:=false;
end;
```

If you set stopped back to false, the transporter starts to move again. The speed-time graph in this variant resembles that in Fig. 6.49.



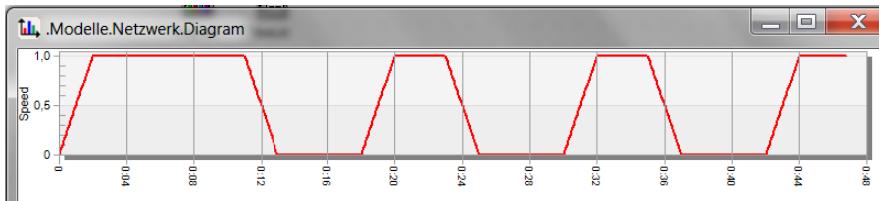
**Fig. 6.49** Speed-time graph

*Variant 2: Decelerate and accelerate*

For this variant, you need a sensor at the position at which the braking starts. In the method, you can wait until the transporter is stopped (movingSpeed = 0) and then run your actions (e.g. loading or unloading). The method could resemble the following:

```
(SensorID : integer; isFront : boolean)
is
do
  @.speed:=0;
  waituntil @.movingSpeed = 0 prio 1;
  wait(5);
  --set speed of the class
  @.speed:=@.class.speed;
  @.stopped:=false;
end;
```

You can see in the diagram that the vehicle is now breaking as well as accelerating (Fig. 6.50).



**Fig. 6.50** Speed-time graph

### 6.7.7 *Drive a Certain Distance*

If you want to model movements more accurately, you may find controls with commands such as "go 1000 mm". Until Version 12 there is no prepared function in Plant Simulation to drive a certain distance with the transporter. You have at least two modeling approaches:

- You start the transporter and check at very short intervals the position of the transporter (e.g. `<transporter>.FrontPos`), e.g. using a generator and a method. If the transporter has reached its position, you stop the transporter (and stop the monitoring with the help of the generator).
- You use a (dynamically set) sensor to stop the transporter at the respective position.

The method a) causes a large number of function calls, which results in a negative influence on the runtime behavior of the simulation. For the variant b) you should simulate a portal loader.

#### Example: Drive a certain distance/portal loader

Create a frame as in Fig. 6.51.

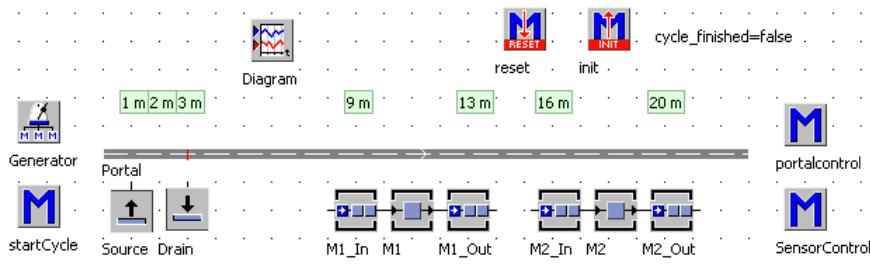


Fig. 6.51 Example frame

Make the following settings: The source creates MUs without any interval. The processing time of M1 and M2 is each two minutes. The buffer M1\_in, M1\_out, M2\_in and M2\_out each have a capacity of one and a processing time of five seconds. Connect the buffers and the SingleProcs with connectors. Create a sensor at the track portal (e.g. on position 20 meters) and connect the sensor with the method SensorControl. The generator calls each second the method startCycle. Set the start of the generator to ten seconds. Insert two user-defined attributes into the portal:

- loadingTime (time; value 15 seconds) for setting the loading and unloading time
- portal\_inPosition (Boolean; value false) for the return message of the portal transporter

Duplicate the vehicle in the class library and rename the duplicate as portaltransporter. Insert into the portaltransporter a user-defined attribute: drive (method). The bases of the modeling are the planned movements of the portal loader. The portal loader operates in a fixed cycle:

- the start of the cycle is a waiting position (absolute two meters), the loader is empty
- the loader moves to the source and loads one part (one meter track backwards  $\rightarrow -1m$ )
- the loader moves with the part to M1\_in and unloads the part (eight meters track forwards)
- the loader moves empty to M1\_out and loads a part, if there is a part (four meters track)
- the loader moves to M2\_in and unloads the part, if he transports a part (three meters track)
- the loader moves to M2\_out and loads a part, if there is a part (four meters track)
- the loader moves to the drain and unloads the part (if the loader is loaded; - 17 meters track)

The movement of the loader will be realized using two user-defined methods:

`<loader>.drive(<distance:real>)` moves the sensor on the track, sets the direction of the loader and starts the loader. If the requested position is outside of the track, an error message is displayed.

`<track>.sensorControl` stops the loader.

To make the movement of the loader observable, we need a user-defined attribute (`<track>. portal_inPosition`). This we set to true if the portal is set in motion (method `drive`) and to false if the portal was stopped (method `sensorControl`).

The method `drive` could resemble the following:

```
(distance:real)
is
do
  --if sensor outside of track --> error message
  if self.~.~.sensorID(1).position+distance < 0
    or self.~.~.sensorID(1).position+distance >
      self.~.~.length then
    Messagebox("The position is located outside of
      the track!",1,3);
  else
    --move sensor
    self.~.~.sensorID(1).position:=
      self.~.~.sensorID(1).position+distance;
    --calculate direction
    if self.~.frontPos >
      self.~.~.sensorID(1).position then
        self.~.~.backwards:=true;
    else
      self.~.~.backwards:=false;
    end;
    --start
    self.~.~.stopped:=false;
    --set portal_inPosition to false
    self.~.~.portal_inPosition:=false;
  end;
end;
```

The method `SensorControl` stops the loader and reports the completion of the movement:

```
(SensorID : integer; Front : boolean)
is
do
  --stop
  @.stopped:=true;
  --report
  portal.portal_inPosition:=true;
end;
```

The method portalControl contains the entire control of the process of the portal: The loader is started. The control will wait until the loader has reached its position. Then it is loaded or unloaded. At the end of the cycle, the method reports the completion by setting the variable cycle\_finished as true:

```
is
do
  tsp:=portal.cont;
  --the cycle of the Portal
  --drive to the source (-1m)
  tsp.drive(-1);
  waituntil portal.portal_inPosition prio 1;
  --load from source
  wait(portal.loadingTime);
  source.cont.move(tsp);
  tsp.drive(8);
  waituntil portal.portal_inPosition prio 1;
  --move the part to M1_in
  wait(portal.loadingTime);
  tsp.cont.move(M1_in);
  --drive to M1_out
  tsp.drive(4);
  waituntil portal.portal_inPosition prio 1;
  --if part on M1_out, then load
  if M1_out.occupied then
    wait(portal.loadingTime);
    M1_out.cont.move(tsp);
  end;
  --drive to M2_in
  tsp.drive(3);
  waituntil portal.portal_inPosition prio 1;
  if tsp.occupied then
    wait(portal.loadingTime);
    tsp.cont.move(M2_in);
  end;
  --drive to M2_out
  tsp.drive(4);
  waituntil portal.portal_inPosition prio 1;
  --if part on M2_out then load
  if M2_out.occupied then
    wait(portal.loadingTime);
    M2_out.cont.move(tsp);
  end;
  --drive to Drain
  tsp.drive(-17);
  waituntil portal.portal_inPosition prio 1;
  if tsp.occupied then
```

```

    wait(portal.loadingTime);
    tsp.cont.move(Drain);
end;
--drive to waiting position and finish job
tsp.drive(-1);
waituntil portal.portal_inPosition prio 1;
cycle_finished:=true;
end;

```

The method portalControl must now be called at an appropriate point in time. This can happen, e.g. using a generator and a method. The method startCycle checks the condition to start a new cycle (old cycle is completed, a new part can be placed on M1\_in etc.) and then calls the method portalControl:

```

is
do
  if cycle_finished and portal.portal_inPosition
    and source.occupied and M1_in.empty then
    cycle_finished:=false;
    portalControl;
  end;
end;

```

For initialization, it is necessary to set the sensor to the waiting position of two meters and to insert the transporter. The generator should have a starting time that allows the transporter to travel to the stop position and stop there (e.g. ten seconds). Therefore, the init method must contain the following instructions:

```

is
do
  --move sensor to 2m
  portal.sensorID(1).position:=2;
  --create portal transporter
  .MUs.Portaltransporter.create(portal,1);
  portal.portal_InPosition:=false;
  diagram.aktiv:=true;
end;

```

#### *Position (distance)-time diagram*

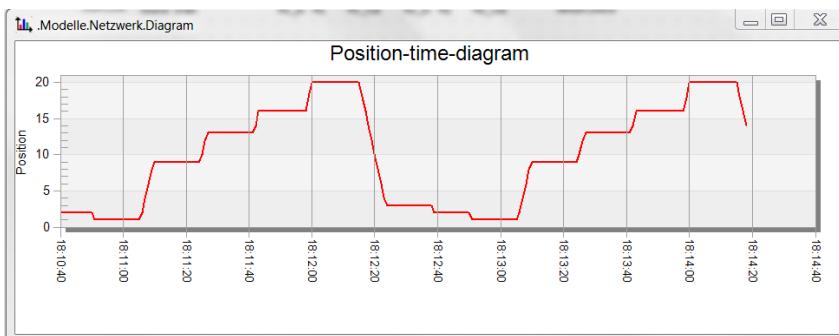
The planning of portal movements is often done in the form of position-time diagrams. We can produce these diagrams without any problem in Plant Simulation. To do this, you can plot the front position of the portal transporter on the track. The following settings are necessary in the diagram block: data input channel: portal.cont.frontPos; mode: sample; interval: one second; display: plotter; axes: range; x: 4:00; number of values: 1.000. At the beginning of the simulation, the portal transporter does not exist; therefore, there is an error message. You can work around this problem by turning off the diagram in the reset method and re-enabling it in the init method after the creation of the transporter. The method reset is as follows:

```

is
do
  diagram.active:=false;
end;

```

The position-time diagram should look like the one in Fig. 6.52.



**Fig. 6.52** Position-time diagram

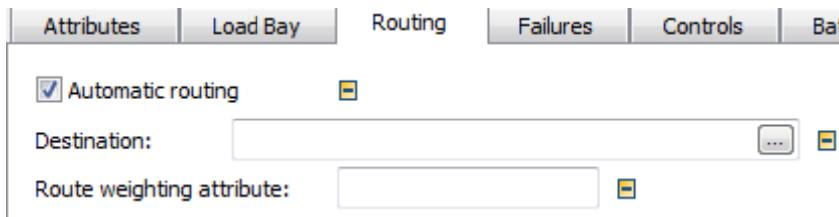
### 6.7.8 Routing

If a track has various successors, different types are available for routing:

- Automatic routing
- Routing with destination lists
- Drive control
- Exit control

#### 6.7.8.1 Automatic Routing

The setting for automatic routing can be found in Plant Simulation Version 11 within an additional tab (Fig. 6.53).

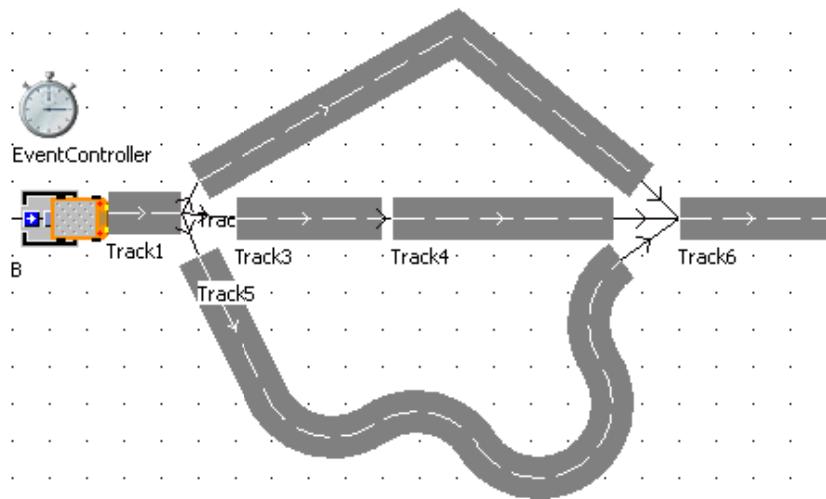


**Fig. 6.53** Routing

**Automatic routing (+Destination):** When you select this option, Plant Simulation searches along the connectors for the shortest route to the destination. All objects on the way to the destination must be connected by connectors.

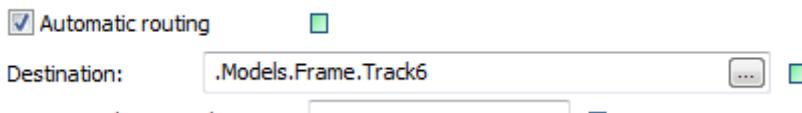
### Example: Automatic routing

Create a frame like in Fig. 6.54.



**Fig. 6.54** Example frame

Drag a transporter from the class library to the buffer. Open the dialog of the transporter by double-clicking it. The destination of the transporter is track6 (Fig. 6.55).



**Fig. 6.55** Routing

Start the simulation and reduce the simulation speed. The transporter finds the shortest way.

**Route weighting:** The automatic routing first evaluates only the lengths of the tracks. Using route weighting, you can control this selection process to take into account additional factors (such as quality of the road, probability of disturbances on the road, etc.). Do this by defining a user-defined attribute on the tracks (data-type real). The transporter reads the attribute that you enter in the field route weighting and multiplies the track lengths with the attribute values. Thereafter, the transporter is relocated to the shortest path. When you set a negative value, then the track is not included in the automatic route search.

### Example: Route weighting

Include in the example above in the track (class library) a user-defined attribute (rw, data-type real, value one). Change the value of this attribute in track3 from one to ten (e.g. to consider a poor quality of the track). Enter in the transporter as route weighting attribute the attribute "rw". The transporter now finds another "shortest" way.

#### 6.7.8.2 Routing (Destination Lists)

This kind of routing does not require connectors. To use automatic routing, you need to supply the transporter with destination information and assign information to the track about which destinations are to be reached on the track (destination list). Plant Simulation searches the destination lists of the successors for the target of the transporter. Plant Simulation then transfers the transporter to the first track whose target list contains the destination.

### Example: Automatic routing with destination lists

A source randomly produces three parts. A transporter loads each one part and transports it to the relevant machine. The transporter then drives back to the source. Each machine can only process one kind of part. A special track leads to each machine. Create a frame according to Fig. 6.56.

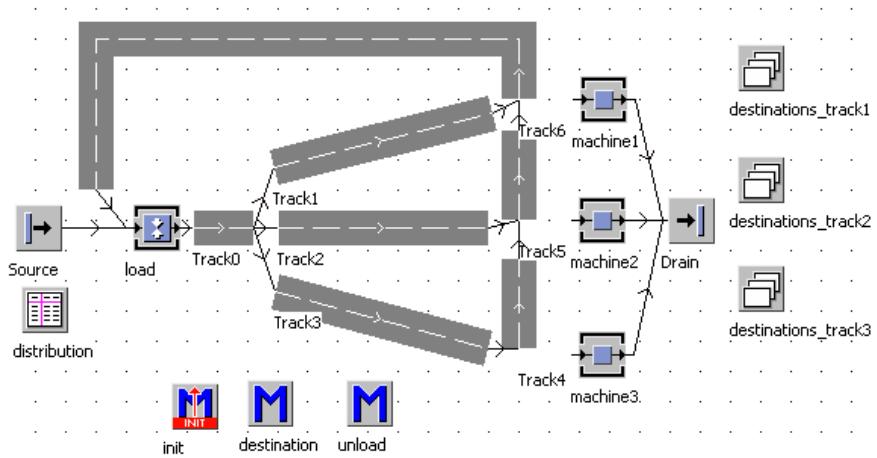


Fig. 6.56 Example frame

1. Duplicate the entity three times. Name the parts as Part1, Part2 and Part3. Color the parts differently.
2. Go to the source. Enter an interval of two minutes. Select MU Selection random. Enter the table distribution into the text box Table (Fig. 6.57).



**Fig. 6.57** Setting source

Plant Simulation formats the table distribution. Open the table. Drag the parts from the class library to the table (this will enter the absolute path into the table). You can also enter the absolute path by yourself. Type the distribution of the parts in relation to the total amount in the second column (Fig. 6.58).

	object	real
string	MU	Frequencies
1	.MUs.part1	0.10
2	.MUs.part2	0.20
3	.MUs.part3	0.70

**Fig. 6.58** Table distribution

The source now produces Part1, Part2 and Part3 in a random sequence.

3. An assembly station loads the transporter.

First, connect track6 with the assembly station, then with the source. Insert track6 so that the exit is located close to the assembly station. Ensure the following settings in the assembly station: assembly mode: attach MUs and assembly table with: predecessors (insert into the assembly table one part from Predecessor 2). The processing time of the assembly station is ten seconds. The init method inserts the transporter close to the exit of track6 into the model. Method init:

```
is
do
  deleteMovable;
  .MUs.Transporter.create(track6, 15);
end;
```

4. Determine the destination of the transporter depending on the name of the part. Set the value of the attribute destination of the transporter with a method. The output sensor (rear) of the assembly station is to trigger the method destination. The method destination sets the attribute depending on the MU names.

```
is
do
  -- @ denotes the transporter
  -- @.cont is the part on the transporter
```

```

if @.cont.name="part1" then
  @.destination:=machine1;
elseif @.cont.name="part2" then
  @.destination:=machine2;
elseif @.cont.name="part3" then
  @.destination:=machine3;
end;
end;

```

5. Create and assign the destination list of the tracks.

To create the destination lists, use objects of type CardFile. The required data-type is object. First, turn off inheritance (Format – Inherit Format). Then click the list header (gray, string) with the right mouse button. Select Format. Select the data-type object on the tab Data type. Now enter the objects that can be reached via the track. You can also enter the destinations by dragging the objects onto the list and dropping them onto the respective line. This inserts the absolute path. Insert Machine1 into the destinations\_track1 and so on. Enter the destination list on the tab Attributes of the track (forward destination list, Fig. 6.59).

Forward destination list:  

**Fig. 6.59** Destination list

6. Move the parts at the end of the tracks. The parts are loaded onto the machines at the end of tracks 1, 2 and 3. To accomplish this, we use the destination addresses of the transporters. Enter the method unload into the exit controls of the tracks 1, 2, and 3 (rear). Method unload:

```

is
do
  -- @ is the transporter
  @.cont.move(@.destination);
end;

```

At the end, the transporter drives by itself to Machine3 on track3 and to Machine2 on track2, etc., depending on which part is loaded.

Note: Plant Simulation transfers the transporter onto the first object in whose destination list the destination of the transporter is registered. If the following track fails, the transporter stops and waits until the failure is removed. While routing, Plant Simulation does not take the status of the tracks into account.

### 6.7.8.3 Routing with SimTalk

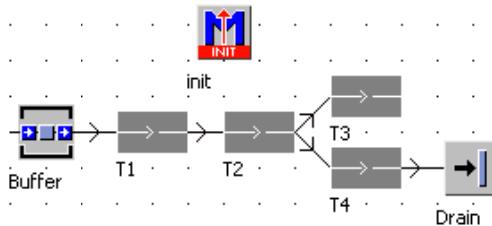
You can determine all successors from an object. If you define a table with all possible destinations in the successor objects, then you can select the relevant successor in an exit control and move the transporter in the right direction. In such a control, you can make various evaluations. You can use the methods from Table 6.3.

**Table 6.3** Methods for determining the successors of an object

Method	Description
<path>.numSucc	Returns the number of successors
<path>.succ([integer])	Returns the successor-object with the index <integer>

#### Example: Routing with SimTalk

Duplicate a track in the class library and insert two user-defined attributes: destinations (table, first-column data-type object) and exitControl (method). Assign to the track the exitControl (front). Create a frame like in Fig. 6.60.



**Fig. 6.60** Example frame

The init method creates a transporter at the buffer and sets the destination-attribute of the transporter to Drain.

```
is
  car:object;
do
  car:=.BES.Fahrzeug.create(buffer);
  car.destination:=drain;
end;
```

Insert Drain into the destinations table of t4. Deactivate in the transporter in the class library the option of Automatic routing. In the exitControl of the tracks, the following should take place: If there is only one successor, then the transporter is moved to the successor. If there are multiple successors, then the method searches through all successors in the destinations tables for the destination of the transporter. If the destination is found, the transporter is relocated to the corresponding successor.

If the destination is not found, an error message is displayed. The exit-control could have the following content:

```

is
  successor:object;
  i:integer;
  found:boolean;
do
  if self.~.numSucc = 1 then
    -- move directly
    successor:=self.~.succ;
    waituntil successor.operational prio 1;
    @.move(successor);
  else
    for i:=1 to self.~.numSucc loop
      self.~.succ(i).destinations.setCursor(1,1);
      -- if found --> move
      if self.~.succ(i).destinations.find(
        @.destination) then
        successor:=self.~.succ(i);
        waituntil successor.operational prio 1;
        @.move(successor);
        return;
      end;
    next;
    messagebox(@.destination.name+
      " is not an entry in the destinations-tables
      of the tracks!",1,3);
  end;
end;

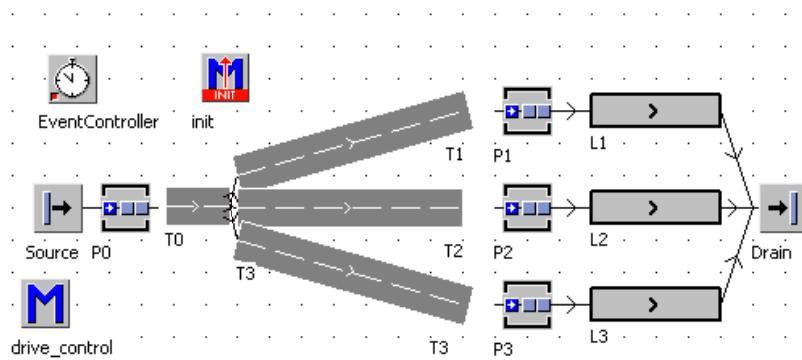
```

#### 6.7.8.4 Driving Control (“freestyle”)

At a junction, you can determine the destination of the transporter with SimTalk, e.g. depending on the availability and the load of the target station, and transfer the transporter on the correct track.

##### Example: Driving control

We want to simulate a manipulation robot (e.g. FlexPicker by ABB). The robot can freely transport parts within a restricted area at high speed. You are to simulate the following problem. The robot takes parts from one place and distributes them onto three lines. The lines have an availability of 98 per cent and an MTTR of 25 minutes. The robot itself has an availability of 99 per cent and an MTTR of 30 minutes. The robot has a speed of 10 m/s (no acceleration). The cycle time is 1.05 seconds (source interval). The speed of the lines is 0.1 m/s. The part has a length of 0.3 meter. The robot has a work area with a diameter of 1.2 meters. Set the scaling factor in the frame to 0.005. Create a frame according to Fig. 6.61.



**Fig. 6.61** Example frame

Length of the tracks T0: 0.2 meter; W1: 0.75 meter; W2: 0.7 meter; W3: 0.75 meter; processing time P1, P2, P3: three seconds (to secure a distance between the parts); the capacity of all buffers is one part. The length of the transporter is 0.1 meter (booking point 0). The transporter must drive backwards after being inserted into the frame. Therefore, select Backwards in the dialog of the transporter in the class library. Proceed as follows:

1. Program the init method. It creates a transporter on the track T0. While creating this transporter, a length is passed so that the transporter can trigger a backward exit sensor.

```
is
do
  .MUs.Transporter.create(T0, 0.1);
end;
```

2. Program the backward exit control drive\_control: The transporter waits until a part is located on P0 and loads it. The transporter drives forward until the end of track T0. A single method is to be used for all controls. For that reason, the object and possibly the direction of the transporter will be examined in the method.

Method drive\_control:

```
is
do
  if ? = T0 and @.backwards then
    -----
    -- T0 exit backwards
    waituntil P0.occupied prio 1;
    P0.cont.move(@);
    @.backwards:=false;
    -----
  end;
end;
```

Assign the method `drive_control` to the track `T0` as the exit control and the backward exit control.

3. Program the exit control `T0`: At the end of the track `T0`, decide on to which place the transporter must drive. The transporter waits until `P1`, `P2`, or `P3` is empty. Starting with the station `P1`, the method queries whether the place is empty. The transporter will be transferred onto the track to the first empty place.

Method `drive_control`: insert a new branch in the query if `? = T0` and `@.backwards` then...:

```

is
do
  if ? = T0 and @.backwards then
  -----
  -- see above
  -----
  elseif ?=T0 and @.backwards= false then
  -- T0 exit
  waituntil P1.empty or P2.empty or
  P3.empty prio 1;
  --drive to the empty place
  if P1.empty then
    @.move(T1);
  elseif P2.empty then
    @.move(T2);
  elseif P3.empty then
    @.move(T3);
  end;
  end;
end;

```

4. Program the exit control of the tracks `T1`, `T2` and `T3`: The transporter loads the part into the buffer. After this, the transporter moves backwards to load a new part. To simplify, define the attribute `buffer` (type `object`) in the class `track` in the class library and assign the buffer `P1` to the track `T1`, etc. (Fig. 6.62).

Name	Value	Type	C.
buffer	P1	object	

**Fig. 6.62** Buffer attribute

Only control is required for unloading. Therefore, you can program it as an else-block in the query of the objects.

```

is
do
  if ? = T0 and @.backwards then
  -----
  -- T0 backwards exit
  -- see above
  -----
  elseif ?=T0 and @.backwards= false then
  -- T0 exit
  -- see above
  else
    --unload onto buffer
    @.cont.move(??.buffer);
    @.backwards:=true;
  end;
end;

```

### 6.7.9 Sensorposition, Sensor-ID, Direction

There are two methods for accessing the sensors of a track, a line and a transporter in SimTalk:

`<path>.sensorID(<integer>)`, using the sensor-id (which is given by Plant Simulation)

`<path>.sensorNo(<integer>)`, by a collection of sensors. The number of sensors you get with `<path>.numSensors`. The sensor itself has attributes according to Table 6.4.

**Table 6.4** SimTalk attributes of the sensor

Attribute	Description
<code>&lt;sensor&gt;.position</code>	Returns the position of the sensor for the position type relative to the percentage based on the length of the block, for the position type length, a length indication on the block
<code>&lt;sensor&gt;.front</code>	Indicates if the option front of the sensor is selected
<code>&lt;sensor&gt;.rear</code>	Indicates if the option rear of the sensor is selected
<code>&lt;sensor&gt;.positionType</code>	Sets/reads the type of the sensor; possible values are “Length” and “Relative”

#### Example: Queuing

The first step is to develop a method by which you can determine the value for the backward movement of a transporter by evaluating two sensor IDs of a track.

These arguments have to be passed:

- track
- current sensorID
- target sensorID

The return value of the function is of the type Boolean. Create the method (getDirection) in the current example. Using `<track>.SensorID(id)` you can access all the information that is associated with a sensor. The method `SensorID` returns an object of type `sensor`. The attribute `position` returns the position of the sensor. To calculate and return the necessary value of the attribute `backwards`, the method needs the following content.

```
(track:object;sensorFrom:integer;sensorTo:integer)
: Boolean
is
  posFrom:real;
  posTo:real;
  backwards:boolean;
do
  posFrom:=track.sensorID(sensorFrom).position;
  posTo:=track.sensorID(sensorTo).position;
  backwards:=(posFrom>posTo);
  return backwards;
end;
```

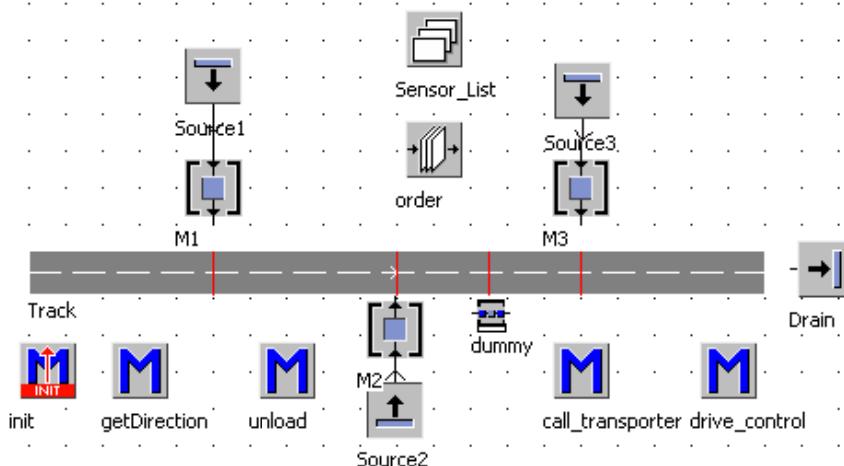


Fig. 6.63 Example frame

A transporter unloads three machines. It waits in its waiting position (12.5 meters) until a driving order arrives. The transporter then drives to the machine and unloads the part. The transporter drives with the part to the end of the track and unloads the part there. Create the frame from Fig. 6.64.

Create sensors on the track (length 20 meters, Fig. 6.64).

ID	Position	Front	Rear	L.	Path
1	5m	x			drive_control
2	10m	x			drive_control
3	12.5m	x			drive_control
4	15m	x			drive_control

**Fig. 6.64** Sensors

Settings: Source1, Source2 and Source3: non-blocking with an interval of one minute; M1, M2, and M3: one minute processing time, availability of 50 per cent, two minutes MTTR, select different random number streams for the different machines (before Version 11); transporter speed: one m/s; capacity: one part; drain: zero seconds processing time. The transporter has a user-defined attribute targetSensorID (integer). The start value is three.

1. The init method creates the transporter on the track. The transporter is to stop at the sensorID 3. The method init appears as follows:

```
is
do
  .MUS.Transporter.create(track,1);
end;
```

The transporter should always stop when the SensorID of the track matches the targetSensorID of the transporter. Program the method drive\_control: (sensorID : integer)

```
is
do
  if sensorID = @.targetSensorID then
    @.stopped:=true;
  end;
end;
```

2. The machines call the transporter after having processed the parts. In this example, the machine should enter its sensorID into a QueueFile. The assignment of machines to the sensors is entered into a Cardfile (sensor\_list). If a sensor is not to be used, enter the object dummy instead (as a wildcard). You cannot leave empty rows in a Cardfile. Also add a dummy object to the frame; otherwise, you will receive an error message. In our example, the sensor list appears as in Fig. 6.65.

	object
1	M1
2	M2
3	dummy
4	M3

**Fig. 6.65** CardFile sensor list

If a part is completed on a machine, it triggers the exit sensor. The machine then has to search the SensorID in the sensor\_list and enter the SensorID into the QueueFile order. Program the method call\_transporter as the exit control of the machines M1, M2 and M3. Select the data-type integer for the QueueFile (order). The method call\_transporter could appear as follows:

```
is
  sensorID:integer;
do
  -- search sensorID
  -- set the position of the cursor to the beginning
  sensor_list.setCursor(1);
  -- search for the machine
  sensor_list.find(?);
  -- position the cursor
  sensorID:=sensor_list.cursor;
  -- insert into order
  order.push(sensorID);
end;
```

3. The transporter is waiting at the waiting position until an order arrives. It then drives to the machine. Using the attribute dim, you can determine the number of entries in a list. This attribute is observable; it can be monitored with an observer or a Waituntil statement. The direction can be determined using the sensor IDs. In this case, this is easy because the sensors are not in a mixed order. If you insert a new sensor afterward (e.g., for a new machine), the sensor IDs get mixed up, which means that a greater number of sensor IDs do not necessarily mean a greater length of the position. Extend the method drive\_control as follows.

```
(sensorID : integer)
is
do
  if sensorID = @.targetSensorID then
    @.stopped:=true;
    if sensorID= 3 then
      waituntil order.dim > 0 prio 1;
      @.targetSensorID:=order.pop;
      @.backwards:= getDirection(?,3,
      @.targetSensorID);
      @.stopped:=false;
    end;
  end;
end;
```

4. The transporter gets an order, drives off, and stops in front of a machine. The transporter has to load the part from the machine and drive forward to the sink. The transporter does not yet know the machine; the method must read the machine

from the sensor list. This is accomplished with the method `read(id)`. Program the method `drive_control` in addition to the example above.

```
(sensorID : integer)
is
do
  if sensorID = @.targetSensorID then
    @.stopped:=true;
    if sensorID= 3 then
      waituntil order.dim > 0 prio 1;
      @.targetSensorID:=order.pop;
      @.backwards:=getDirection(?,3,
      @.targetSensorID);
      @.stopped:=false;
    else
      sensor_list.read(sensorID).cont.move(@);
      @.backwards:=false;
      @.stopped:=false;
    end;
  end;
end;
```

5. Program the method `unload` for unloading the part onto the drain, and assign it as the exit control of the track. The transporter must unload the part to the drain. The transporter then drives forward. If an order exists, it must be read and the `targetSensorID` of the transporter must be set anew. If no order exists, then the `targetSensorID` is three. Program the method `unload` (exit control track):

```
is
do
  @.stopped:=true;
  @.cont.move(drain);
  if order.dim > 0 then
    -- to machine
    @.targetSensorID:=order.pop;
  else
    -- to waiting position
    @.targetSensorID:=3;
  end;
  @.backwards:=true;
  @.stopped:=false;
end
```

### 6.7.10 Start Delay Duration

A transporter stops automatically when it collides with a standing transporter on the same track. This starts the first transporter again and then all the collided

transporters automatically start again. To model this behavior more realistically, you can use the attribute Start delay duration (e.g., 0.5 seconds). The following transporter will start with a lag of 0.5 seconds after the start of the transporter in front. You can set the start delay duration on the dialog of the transporter (class library, Fig. 6.66).



Fig. 6.66 Start delay duration

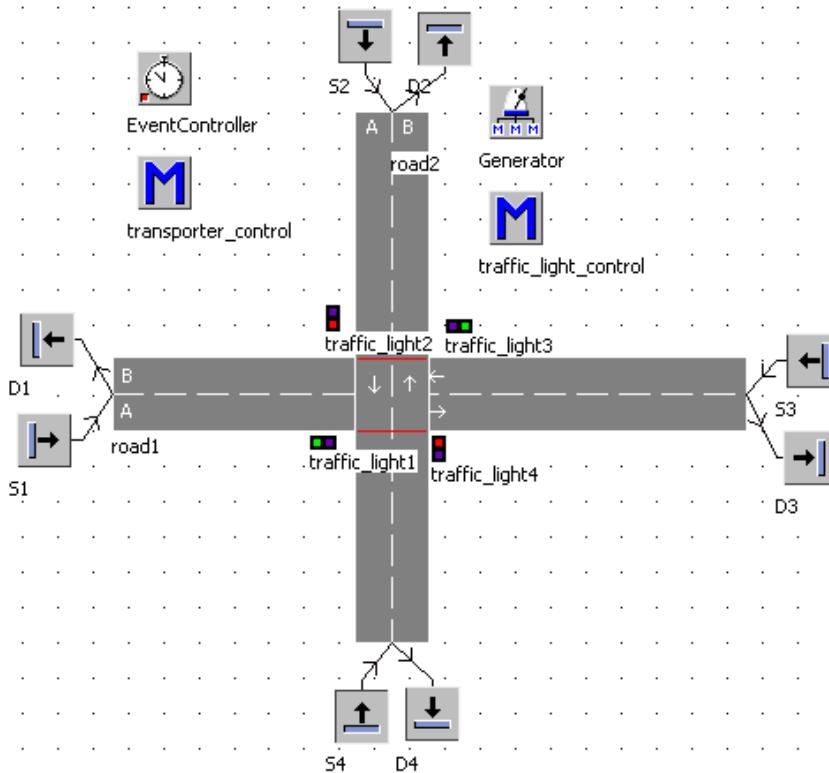


Fig. 6.67 Example frame

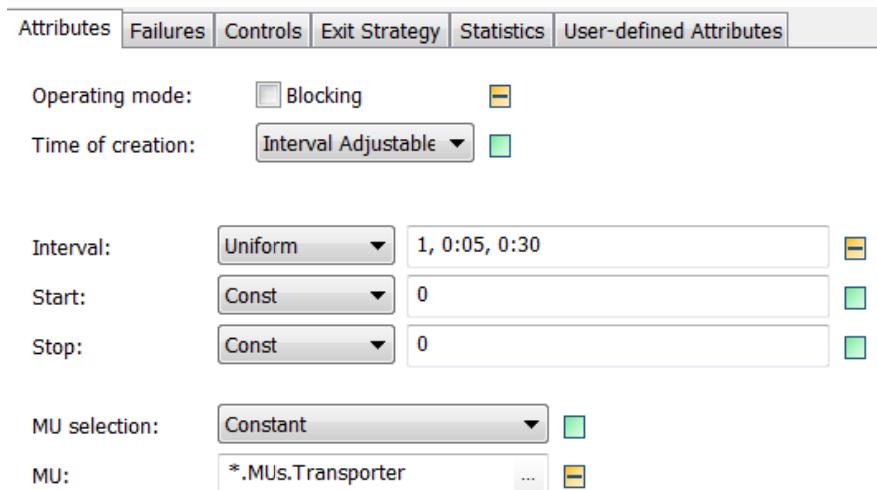
#### Example: Start delay duration, crossroads

You are to simulate a simple crossroads. The crossroads is regulated by traffic lights. In this example, the transporter decides directly at the crossroads whether it stops or carries on (without slowing down). All transporters behind it drive against it. Insert a traffic\_light in the class library (duplicate a class SingleProc and rename it). Create two icons in the class traffic\_light (icon1: green; icon 2: red). Insert in the class traffic\_light a user-defined attribute “go” (data-type Boolean).

Create a new frame. Set the scaling factor to 0.25 (Frame window – Tools – Scaling factor). Set up the frame from Fig. 6.67.

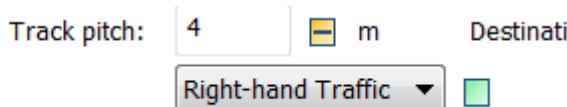
In this example, we also show how the TwoLaneTrack works.

Settings: The sources S1, S2, and S3 create each transporter. The interval of the creation should be randomly distributed. Recreate the setting from Fig. 6.68 in the source S1.



**Fig. 6.68** Settings of the source S1

Recreate this setting also for S2, S3 and S4. Set the stream (first number in field interval for each source to another value). The transporters move at a speed of ten meters per second and accelerate with 10 m/s<sup>2</sup>. To ensure that the transporters can pass each other, set along the paths a track pitch of four meters (Fig. 6.69).



**Fig. 6.69** Track pitch

Create at the crossroads sensors on the lanes. For each lane, you can specify its own sensors. You must uncheck for the other lane the checkboxes for the front and rear. For every track, you need to specify two sensors: one sensor in Lane A and one sensor in Lane B. All sensors will trigger the method transporter\_control. In case of road1, it could look appear in Fig. 6.70.

ID	Position	Front	Rear	L.	Path
1	35m / 55m	A			transporter_control
2	45m / 45m	B			transporter_control

**Fig. 6.70** Sensors

Initialize the simulation, so that two traffic lights show the icon 2 (red) (right mouse button – next icon) and the attribute go is false; the other two traffic lights show the green icon and the attribute go has the value true.

#### *Traffic Light Control*

For traffic light control, we use in this example a method (traffic\_light\_control) and a generator. The method switches the lights and the generator repeatedly calls the method at an interval of one-and-a-half minutes. Method traffic\_light\_control:

```

is
do
  --switches the traffic light
  -- icon1 green, icon2 red
  if traffic_light1.go then
    traffic_light1.go:=false;
    traffic_light1.CurrIconNo:=2;
    traffic_light3.go:=false;
    traffic_light3.CurrIconNo:=2;

    traffic_light2.go:=true;
    traffic_light2.CurrIconNo:=1;
    traffic_light4.go:=true;
    traffic_light4.CurrIconNo:=1;
  else
    traffic_light1.go:=true;
    traffic_light1.CurrIconNo:=1;
    traffic_light3.go:=true;
    traffic_light3.CurrIconNo:=1;

    traffic_light2.go:=false;
    traffic_light2.CurrIconNo:=2;
    traffic_light4.go:=false;
    traffic_light4.CurrIconNo:=2;
  end;
end;

```

You can test the method by starting this repeatedly. The lights should “switch”. The method is called by the generator. Open the generator with a double-click and recreate the settings from Fig. 6.71 on the tab Times.

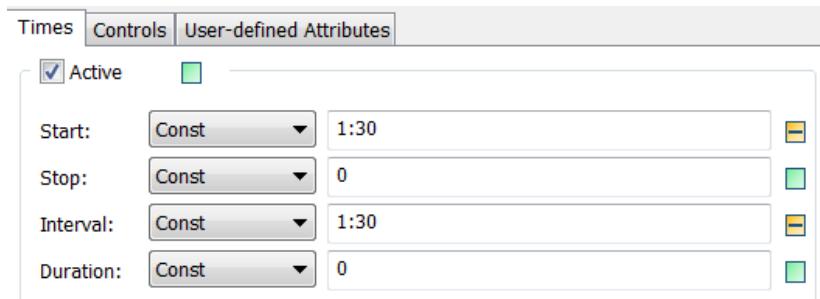


Fig. 6.71 Generator times

Enter the method `traffic_light_control` in the field Interval on the tab Controls (Fig.6.72).



Fig. 6.72 Generator controls

The transporter should stop when the relevant traffic light has the value `go = false` and should wait until `go = true`. Then, the transporter should start again. If Lane A has Sensor 1 and Lane B has Sensor 2, the method `transporter_control` should appear as follows:

```
(sensorID : integer)
is
do
  if ?=road1 and sensorID=1 then
    if traffic_light1.go=false then
      --traffic_light is red - stop
      @.stopped:=true;
      waituntil traffic_light1.go prio 1;
      @.stopped:=false;
    end;
  elseif ?=road1 and sensorID=2 then
    if traffic_light3.go=false then
      @.stopped:=true;
      waituntil traffic_light3.go prio 1;
      @.stopped:=false;
    end;
  elseif ?=road2 and sensorID=1 then
    if traffic_light2.go=false then
      @.stopped:=true;
```

```

waituntil traffic_light2.go prio 1;
@.stopped:=false;
end;
elseif ?=road2 and sensorID=2 then
  if traffic_light4.go=false then
    @.stopped:=true;
    waituntil traffic_light4.go prio 1;
    @.stopped:=false;
  end;
end;
end;

```

Change the start delay duration in the class transporter (class library) to 0.5 seconds and watch what happens.

**Note:** If the transporter does not stop exactly on the line of the sensor, then the reference point of the vehicle is in the wrong position. The reference point is in the default icon in the middle of the symbol. If the transporter is to stop exactly on the line, then you need to set the reference point to the right edge of the icon.

### 6.7.11 Load Bay Type Line, Cross-Sliding Car

For the load bay type line added in Version 10, there are a large number of applications. A variant of these transporters is the so-called cross-sliding car. The cross-sliding car is a transporter that moves forward and back on a track, transversely to the conveying direction of the material flow (usually conveyor lines). The transporter has its own conveyor line (e.g. as a belt conveyor or roller conveyor) by means of which the parts are conveyed to the transporter and from the transporter to the next conveyor section. The transporter has to control these conveyor lines, a number of methods and attributes (selection, Table 6.5):

**Table 6.5** Load bay attributes and methods

Attribute/ Method	Description
<path>.LoadBaySpeed	Speed of the load bay
<path>.LoadBayLength	Length of the load bay
<path>.LoadBayBackwards	Direction of the load bay

The load bay itself can be equipped with sensors as well as entrance and exit controls.

#### **Example: Cross cross-sliding car 1**

You are required to simulate a cross-sliding car which connects four conveyor lines. First, create a frame according to Fig. 6.73.

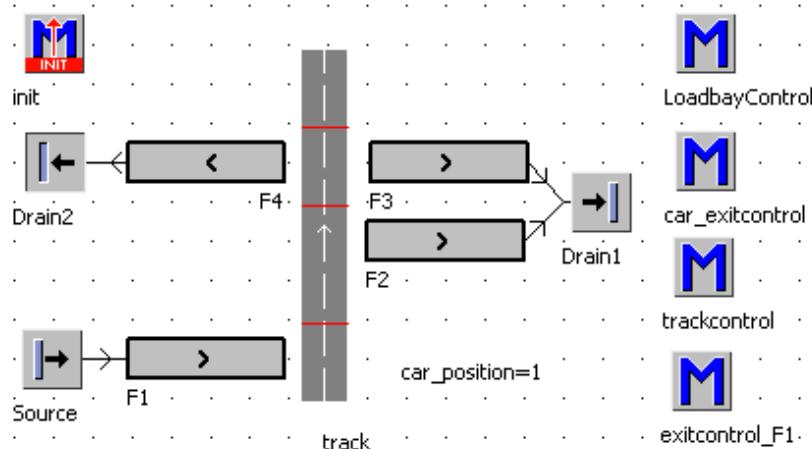


Fig. 6.73 Example frame

Change the setting of the transporter in the class library as follows: Add a user-defined attribute to the transporter (targetSensorID, data type integer, initial value = 1). Select in the tab Load bay, the type “line” (length: 1.5 meters; speed: one m/s; accumulating, no acceleration, backward). Assign the method exitControl\_car to the exit control and the backward exit control of the transporter load bay using the button Controls. Use the button Sensors for inserting a sensor at position 1.25 meters, and assign to the sensor the method loadbayControl. The source creates one part each minute. The line F1 has as its exit control (front) method exitcontrol\_F1. The track needs three sensors (on the positions of the lines) and the sensor control is trackcontrol. Insert in the entity in the class library a user-defined attribute: target (data-type object). You are required to simulate the following sequence: If a part arrives at the output of F1, then the target is set in the part (F2, F3, F4. statistical distribution 10, 20, 70 per cent). Then the part is waiting until the vehicle is at position 1 (in front of F1). The part is relocated to the transporter and conveyed from there to the middle of the transporter. In the middle of the transporter, the line is stopped and the transporter itself is set in motion. At the right sensor position, the transporter is stopped and the line on the transporter is started in the right direction. At the end of the load bay, the part is relocated to the following conveyor section.

Part 1: init method.

In the init method, the transporter is created on the track (near the position of Sensor 1).

```

is
do
  .MUS.car.create(track, 2);
end;

```

Part 2: Exit-control F1:

```

is
  randValue:real;
do
  -- wait for car
  waituntil car_position = 1 prio 1;
  -- distribute targets 10%, 20%, 70%
  randValue:=z_uniform(1,0,1);
  if randValue>=0 and randValue < 0.1 then
    @.target:=F2;
    track.cont.targetSensorID:=2;
  elseif randValue>=0.1 and randValue < 0.3 then
    @.target:=F3;
    track.cont.targetSensorID:=3;
  else
    @.target:=F4;
    track.cont.targetSensorID:=3;
  end;
  -- move to car
  @.move(track.cont);
end;

```

### Part 3: load-bay control

```

(SensorID : integer; Rear : boolean)
is
do
  --set speed to 0
  track.cont.loadBaySpeed:=0;
  --move car
  track.cont.backwards:=false;
  track.cont.stopped:=false;
  car_position:=-1;
end;

```

### Part 4: Track-control

```

(SensorID : integer; Rear : boolean)
is
do
  if sensorID = @.targetSensorID then
    @.stopped:=true;
    --set position
    car_position:=sensorID;
    -- set direction of the loadBay
    if sensorID=1 then
      @.loadBayBackwards:=false;
    elseif sensorID=2 then
      @.loadBayBackwards:=false;
    elseif sensorID=3 and @.cont.target=F3 then

```

```

    @.loadBayBackwards:=false;
elseif sensorID=3 and @.cont.target=F4 then
    @.loadBayBackwards:=true;
end;
--start loadbay of the transporter
@.loadBaySpeed:=1;
end;
end;

```

#### Part 5: car-exit-control

```

(obj : object)
is
do
    --unload the transporter
    if track.cont.targetSensorID =2 then
        @.move(F2);
    elseif ?.LoadBayBackwards then
        @.move(F4);
    else
        @.move(F3);
    end;
    waituntil ?.empty prio 1;
    -- drive to the loading position
    ?.targetSensorID:=1;
    ?.backwards:=true;
    ?.stopped:=false;
end;

```

To simplify the modeling of cross-sliding cars, Plant Simulation provides a tool starting from Version 10. The cross-sliding car was developed as a user object and is included by default in the class library. Select File and then Manage Class Library. Select the tool and click Apply. You can then find the cross-sliding car in the class library within the folder Tools. The cross-sliding car is based, like the example above, on the track block in conjunction with a transporter with a load bay type line. For simple applications, you can easily create an appropriate model with this tool.

#### Example: Cross cross-sliding car 2

Create a frame according to Fig. 6.74 using the Plant Simulation cross-sliding car object.

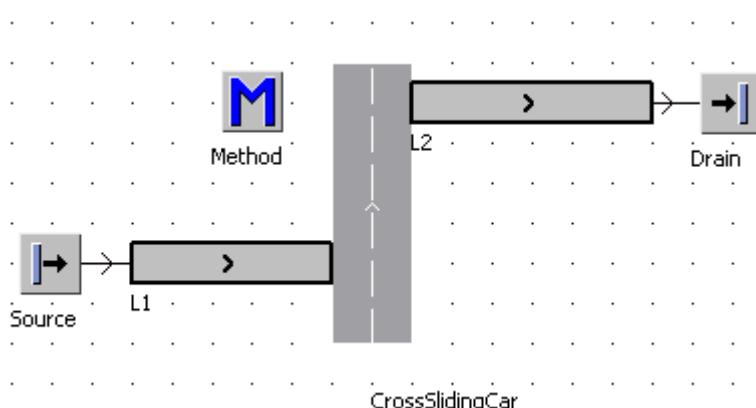


Fig. 6.74 Example frame

The easiest way to connect the lines with the cross-sliding car is as follows: Drag the line on the track of the cross-sliding car until the mouse pointer shows the linking icon (repeat this for each line section). To control and customize this, you can open the list of attached lines in the tab Track Attributes in the dialog box of the cross-sliding car (Fig. 6.75).

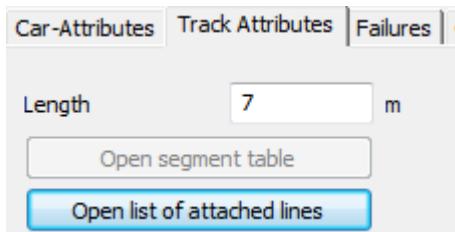


Fig. 6.75 Attached lines

In the table "List of attached lines", you can set the assigned line as well as determine its position relative to the shuttle transporter and the sensor position more accurately (Fig. 6.76).

	Object	Side	CSC Sensor Position	CSC Sensor Id
1	*.Modelle.Netzwerk.L1	L	2	1
2	*.Modelle.Netzwerk.L2	R	6	2

Fig. 6.76 List of attached lines

The cross-sliding car uses the attribute Destination of the transported blocks for the determination of the destination of the transport. You must set this attribute

before the MU is moved to the shuttle transporter. Assign the method to the entrance control of the line L1. Insert the following programming in the method:

```
is
do
  @.destination:=L2;
end;
```

The cross-sliding car should now work.

## 6.8 Tractor

With the object tractor (transporter with the attribute `isTractor = true`), you can easily simulate trains that consist of a tractor and some attached transporter objects (e.g. wagons).

### 6.8.1 General Behavior

On the tractor, you can hitch and unhitch transporters. You have access within the train to all transporters (wagons), e.g. to load and unload them. Practically speaking, the coupled transporters adjust their speed to the tractor. Plant Simulation provides a suite of commands for the simulation of "trains" (Table 6.6).

**Table 6.6** SimTalk attributes and methods of the tractor

Attribute/ Method	Description
<code>&lt;transporter&gt;.hitchFront</code>	Couples the next transporter on the train (front or rear)
<code>&lt;transporter&gt;.hitchRear</code>	The transporter in front or behind the transporter will be unhitched
<code>&lt;transporter&gt;.unhitchFront</code>	Returns the tractor of the train
<code>&lt;transporter&gt;.unhitchRear</code>	Using the methods makes it possible determine a reference to the wagon/tractor in front or behind a wagon/tractor
<code>&lt;transporter&gt;.getTractor</code>	Activates/deactivates the tractor function of the transporter
<code>&lt;transporter&gt;.getFrontWagon</code>	Returns the nearest MU on the same object (e.g. track or line)
<code>&lt;transporter&gt;.getRearWagon</code>	
<code>&lt;transporter&gt;.isTractor</code>	
<code>&lt;mu&gt;.frontMU</code>	
<code>&lt;mu&gt;.rearMU</code>	

### 6.8.2 Hitch Wagons to the Tractor

You can hitch wagons using the methods `hitchFront` and `hitchRear`. For hitching, you can use the collision control of the transporter. The transporter behind the tractor can be accessed via `@.rearMu`. The method of hitching could appear in the simulation as follows: The tractor drives backward against a transporter (wagon). This triggers the collision control. In the collision control, the `rearMU` is hitched to

the tractor/wagon. After a short waiting time, the tractor (with the hitched wagons) starts to move forward again.

### Example: Hitch tractor

In the first example, the tractor has to hitch one wagon. Create a frame according to Fig. 6.77.

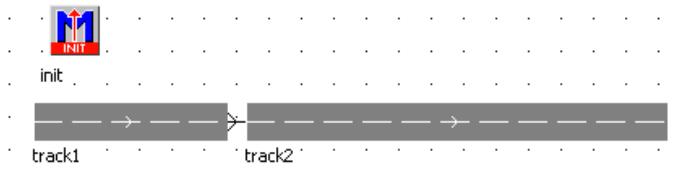


Fig. 6.77 Example frame

The length of track1 needs to be at least six meters. Duplicate the transporter in the class library and rename the duplicate to “tractor”. Activate the checkbox Is tractor and Backwards. Create in the tractor the following user-defined attributes:

- collisionControl (method)
- hitchTime (time; ten seconds)
- num\_wagons (integer; two)

Enter in the tab Control the collisionControl in the field Collision. We will program the collision control as an internal control of the tractor. This implies some special features. The big advantage is that we have to program this method only once for all tractors. If we create a new tractor from the class library, then the method is inherited to this instance. The access to the tractor within the internal method is ensured via `self.~`. In this case, `self` refers to the method and the tilde (~), the overlying object (location, the tractor). The collision control of the tractor could appear as follows:

```

is
  i:integer;
  car:object;
do
  -- hitch wagons behind the tractor
  -- move after hitch-time
  if self.~.backwards then
    car:=self.~;
    for i:=1 to self.~.num_wagons loop
      -- is there a wagon?
      if not isVoid(car.rearMU) then
        car.rearMU.hitchFront;
        car:=car.rearMU;
      else
        exitloop;
    end
  end
end

```

```

    end;
next;
wait(self.~.hitchTime);
self.~.backwards:=false;
self.~.stopped:=false;
end;
end;

```

Next, set the transporter in the class library to backward. For testing, we will create a few transporters on track1 and a tractor on track2. If all transporters move backward, the transporter should "wait" for the tractor. Enter the following lines in the init method:

```

is
do
  --wagons
  .MUS.transporter.create(track1, 1.6);
  .MUS.transporter.create(track1, 3.2);
  .MUS.transporter.create(track1, 4.8);
  -- tractor
  .MUS.tractor.create(track2);
end;

```

If you now start the simulation, the tractor will drive backward against the standing transporters, hitch one and then move forward to the end of the track. With a few changes in the collision control, we can cause the tractor to hitch a certain number of transporters (wagons). For this, we define in a loop the last wagon in the train as the current transporter and hitch this to the next transporter (behind) until we reach the required number. We need to check in this approach whether another transporter exists at the end of the train and can be hitched. CollisionControl:

```

is
  i:integer;
  car:object;
do
  -- hitch wagons behind the tractor
  -- move after hitch-time
  if self.~.backwards then
    car:=self.~;
    for i:=1 to self.~.num_wagons loop
      -- is there a wagon?
      if not isVoid(car.rearMU) then
        car.rearMU.hitchFront;
        car:=car.rearMU;
      else
        exitloop;
      end;
    next;
  end;

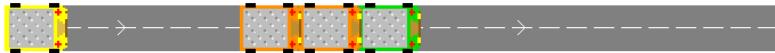
```

```

wait(self.~.hitchTime);
self.~.backwards:=false;
self.~.stopped:=false;
end;
end;

```

Now change the value of the attribute num\_wagons in the tractor in the class library and start the simulation. The tractor now hitches a small train with the adjusted number of wagons (Fig. 6.78).



**Fig. 6.78 Train**

### 6.8.3 *Loading and Unloading of Trains*

There are different ways to load a train:

- The train stops at a position and the individual wagons are loaded while the train is waiting at this position (the loading position changes)
- The train has to move after the loading of a wagon at one position (wagon-length) because the loading position is fixed

Both variants also exist for unloading. In industrial practice, there is a further variant: The wagon is not unloaded immediately, but is unhitched and moved to the point of use.

#### **Example: Unload trains complete**

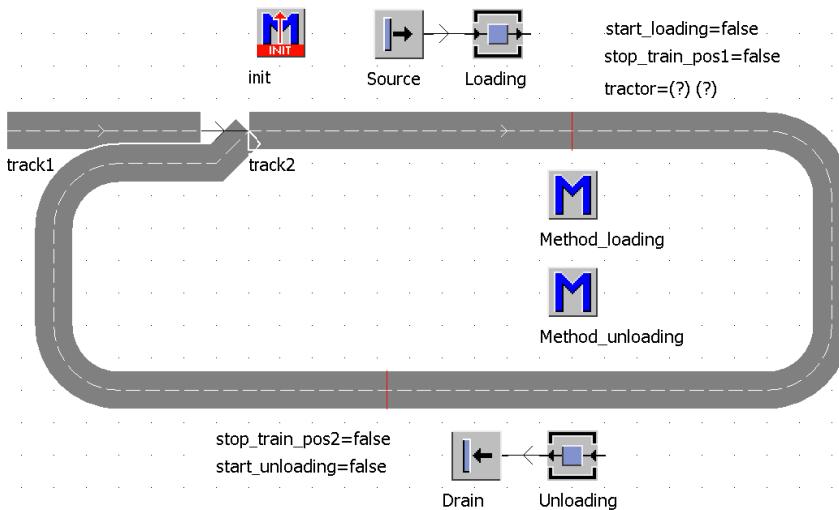
You have to simulate the following procedure: A train arrives at a loading station and is loaded there with three MUs. The train then drives to a second position and is unloaded completely there. Signals control the time at which the train is allowed to drive again. At the beginning of the simulation, the tractor must first pick up the individual wagons. Use the example “Hitch tractor” as a base and create a frame according to Fig. 6.79.

Do not connect the end of the track2 with its beginning (otherwise, we would hinder the hitching the wagons). Instead, add a user-defined attribute in track2 (exitControl, data-type method). Assign the method exitControl in the tab Control to the exit control. Write the following commands in the method:

```

is
do
  @.move (?);
end;

```



**Fig. 6.79** Example frame

Thus, the train is moved back to the entrance of the same track and is able to drive "in a closed circle". Define in track2 a second user-defined method (sensorControl). Insert two sensors in track2 and assign the method sensorControl to the sensors. The method\_loading is connected by an observer with the variable start\_loading, whereas the method\_unloading is connected with the variable start\_unloading. Ensure the following settings: loading: processing time of 30 seconds, source of 30 seconds interval, non-blocking; unloading: processing time of 30 seconds. The variables have start values, as you can see in Fig. 6.79. Set the user-defined attribute num\_wagons of the tractor to three. Change the init method so that the tractor is created at three meters on track2.

As a first step, the train should be stopped at the sensors. The stop\_train\_pos1/stop\_train\_pos2 variables are set to true and start\_loading/start\_unloading also to true. The loading and unloading we will program in the methods Method\_loading and Method\_unloading. For loading and unloading, we need a reference to the tractor of the train. Since we have several vehicles on the same track or even several trains on one track, you must provide a reference to the respective tractor for the methods. If you have several trains on the same track, you will need a separate variable for the tractor for each sensor. You can stop only the tractor in a train. So we need to check with isTractor whether the car that triggers the sensor control is the tractor (the individual hitched wagons also trigger the sensor control). The sensorControl of track2 could appear as follows:

```
(SensorID : integer; Rear : boolean)
is
do
  -- stop the sensors for both sensors
```

```
-- react only to the tractor
if @.isTractor then
    tractor:=@;
    @.stopped:=true;
    if sensorID= 1 then
        stop_train_pos1:=true;
        start_loading:=true;
        waituntil stop_train_pos1 = false prio 1;
    elseif sensorID =2 then
        stop_train_pos2:=true;
        start_unloading:=true;
        waituntil stop_train_pos2 = false prio 1;
    end;
    @.stopped:=false;
end;
end;
```

Starting from the tractor, we can access all the hitched wagons. There are two options:

- getRearMU
- getRearWagon

If these methods return void, then the current object is the last wagon in the train. The loading of the train could proceed as follows: The loading is carried out with the help of a loop. We get the quantity of wagons through the user-defined attribute num\_wagons. You will have to wait until a part for loading is available and the loading time is over (the loading station no longer works). The part is then loaded onto the actual wagon. If the train is fully loaded, we set the variable stop\_train\_pos1 to false (the train should start moving again).

Method\_loading:

```
(Attribut: string; oldValue: any)
is
    i:integer;
    wagon:object;
do
    if start_loading=true then
        wagon:=tractor;
        for i:=1 to wagon.num_wagons loop
            wagon:=wagon.getRearWagon;
            waituntil loading.occupied and
            loading.resWorking=false prio 1;
            loading.cont.move(wagon);
        next;
        --train is full
        stop_train_pos1:=false;
        start_loading:=false;
```

```
    end;
end;
```

Similarly, it could look like the method for unloading the train. We wait in the loop until the unloading place is free, then the content of the current wagon is moved to the unloading place. If the train is empty, we set the variable `stop_train_pos2` to false and the train continues to drive. Method `_unloading`:

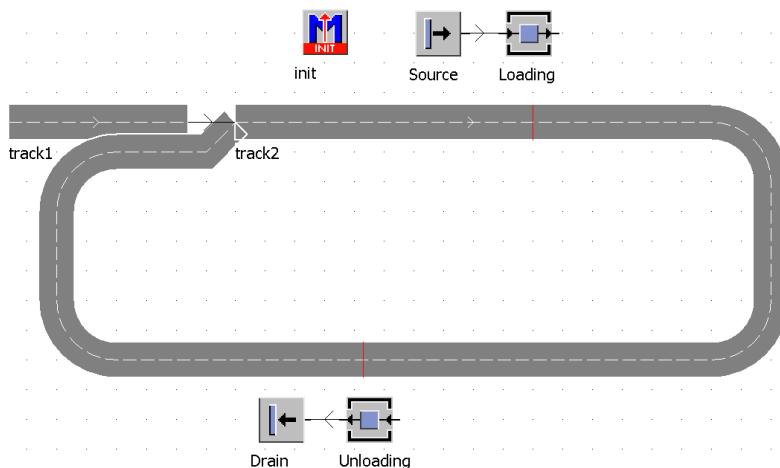
```
(Attribut: string; oldValue: any)
is
  i:integer;
  wagon:object;
do
  if start_unloading then
    wagon:=tractor;
    for i:=1 to tractor.num_wagons loop
      wagon:=wagon.getRearWagon;
      waituntil unloading.empty prio 1;
      wagon.cont.move(unloading);
    next;
    --train is empty
    stop_train_pos2:=false;
    start_unloading:=false;
  end;
end;
```

The loading of the wagons of the train at a fixed position is easier to implement. You create a sensor each for the loading and unloading. At this sensor, the wagon stops the tractor. The associated tractor can be determined using the method `getTractor`. With `isTractor` you determine whether the wagon (@) is the tractor. In the sensor control, we can then program the loading and unloading.

### Example: Load/unload the train wagon by wagon

Use the frame from the example “Unload trains complete” and change it as in Fig. 6.80.

At Position 1, the train should stop by stopping the tractor when the transporter (@) is not the tractor. Then, we wait until the loading is occupied and the processing time is over and load the MU on the wagon. We then start the tractor again. At the unloading position, we proceed similarly, except that we wait for unloading to be empty and then move the MU from the wagon to the station unloading. The `sensorControl` of `track2` could appear as follows:

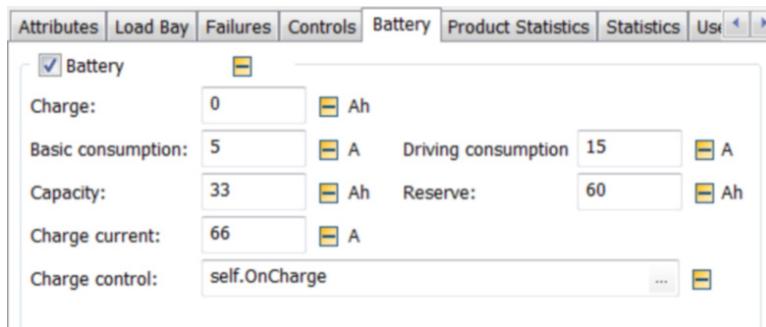


**Fig. 6.80** Example frame

```
(SensorID : integer; isFront : boolean)
is
do
  -- let pass the tractor
  if not @.isTractor then
    -- in a train only the tractor can stop
    @.getTractor.stopped:=true;
    if sensorID= 1 then
      waituntil loading.occupied and
        loading.resWorking=false prio 1;
      loading.cont.move(@);
    elseif sensorID =2 then
      waituntil unloading.empty prio 1;
      @.cont.move(unloading);
    end;
    @.getTractor.stopped:=false;
  end;
end;
```

## 6.9 Model Transporters with Battery

The transporter has properties and methods for modeling the behavior in a battery operation (Fig. 6.81). In the case of battery operation, capacity and consumption data are assigned to the transporter. During the idle phases and the driving of the vehicle, power is consumed. If the battery is completely discharged, the transporter stops. You can define a value when the transporter switches to "reserve". If the reserve capacity is reached, the charge control of the transporter is called. Charging the battery has to be triggered with a method.



**Fig. 6.81** Transporter battery attributes

**Checkbox Battery:** By default, the battery function is disabled. Mark the checkbox to activate the battery function.

**Charge:** Plant Simulation calculates this value on the basis of charge, basic or driving consumption or charge current. Enter here the initial charge of the transporter.

**Basic/Driving Consumption:** Enter the basic consumption of the vehicle in idle status or in motion.

**Capacity:** The nominal capacity of the battery is entered in this field. Plant Simulation charges the battery up to that capacity.

**Reserve:** Batteries are usually not completely discharged. Most have to retain a residual charge of e.g. 20–25 per cent in order to prevent damage to the battery. You must also consider the time required for the vehicle to drive to the place of charging here.

**Charge Current:** The charging current is determined by the type of battery. In the normal case, the charging current is a maximum of 20 per cent of the normal capacity. The charging time is calculated in Plant Simulation linearly based on the charge, capacity and charging current. If, e.g. you have to charge from a charge of 60 Ah to a charge of 330 Ah with a charging current of 66 A, then Plant Simulation takes this into account for  $(330 - 60)/66 = 4.09$  hours.

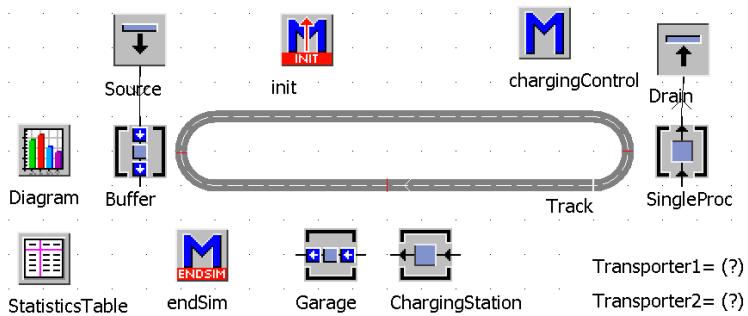
**Charge Control:** The charge control is called by Plant Simulation when the reserve is reached, the charge reaches zero or charging is complete (charge = capacity). The charging of the battery is started by setting the attribute BatCharging to true. Table 6.7 shows the SimTalk attributes and methods control the battery behavior of the transporter:

**Table 6.7** Battery attributes and methods

Method/Attribute	Description
<path>.batCharging	Sets and indicates the battery charge status of the transporter; unfortunately, this attribute is not observable—you must instead use the charge control of the transporter in order to respond to the end of the charging (Boolean)
<path>.batBasicCons	Sets/reads the base power consumption of the transporter (real)
<path>.batDriveCons	Sets/reads the driving power consumption of the transporter (real)
<path>.batCapacity	Sets/reads the nominal capacity (real)
<path>.batChargeCurrent	Sets/reads the charge current (real)
<path>.batCharge	Sets/reads the actual charge (real); if you set the charge to the battery capacity of the transporter, then the charging of the battery is terminated
<path>.batReserve	Sets/reads the reserve of the battery (real)

**Example: Transporter with battery**

A SingleProc is loaded with a transporter. For this, the transporter loads one part and transports it 15 meters to the SingleProc. It then returns empty. The processing time of the SingleProc is 100 seconds. There are two transporters in the simulation; if one transporter is charging the battery, then the other transporter is in use. Create a frame as in Fig. 6.82.

**Fig. 6.82** Example frame

Create three sensors on the track:

- Sensor 1 near the ChargingStation
- Sensor 2 near the buffer
- Sensor 3 near the SingleProc

The source produces entities at intervals of 100 seconds. The buffer and the garage have a capacity of one MU. Ensure the following settings in the transporter in the class library: The transporter has a speed of 0.5 m/s. The battery settings correspond to the settings in Fig. 6.81 (capacity: 330 Ah; reserve: 60 Ah; based consumption: 5A; driving consumption: 15 A; charge current: 66 A). Set the charge at the beginning to 330 Ah (fully charged). Enter as charge control, the method chargingControl and activate the battery function. Create in the init method two transporters (one on the track in front of Sensor 2 and one in the garage):

```
is
do
  transporter1:=.MUs.transporter.create(track,8);
  transporter2:=.MUs.transporter.create(garage);
end;
```

The control at Sensor 2 should have the following content: The transporter stops and waits until a part is present on the buffer. Then, the transporter loads the part and continues to drive (loading time is not taken into account):

```
--sensorControl SensorID 2
@.stopped:=true;
waituntil buffer.occupied prio 1;
buffer.cont.move(@);
@.stopped:=false;
```

At Sensor 3, the transporter waits until the SingleProc is empty, loads the MU to the SingleProc and continues to drive:

```
--sensorControl SensorID=3
@.stopped:=true;
waituntil SingleProc.empty prio 1;
@.cont.move(singleProc);
@.stopped:=false;
```

At Sensor 1, we check whether the battery level has reached the reserve. If it has, then the transporter moves to the charging station and starts loading. At the same time, the other transporter is placed out of the garage on the track:

```
if @.BatCharge <= @.BatReserve then
  @.batCharging:=true;
  @.move(chargingStation);
  garage.cont.move(track,7);
end;
```

In chargingControl, the already-charged transporter (charge = capacity) is moved into the garage. Since the loading control is also called when the reserve is reached, we must first examine whether the charging station is occupied.

```

is
do
  if chargingStation.occupied then
    if chargingStation.cont.batCharge >=
      chargingStation.cont.batCapacity then
        chargingStation.cont.move(garage);
      end;
    end;
  end;
end;

```

Put a breakpoint on the line @.batCharge: = true; in the control of Sensor 1 in order to control the operation better. For statistical evaluation of the battery charging time, Plant Simulation provides the attributes statBatChargePortion and statBatChargeCount.

## 6.10 Case studies

### 6.10.1 The Plant Simulation Multi-Portal Crane Object

You can create cranes with a combination of tracks and transporters. Plant Simulation provides two objects for the modeling of cranes. The StorageCrane is primarily used for the modeling of storage and retrieval in a warehouse area, while the multi-portal crane is a flexible concept for the modeling of transport processes that are handled by portal cranes.

The multi-portal crane object is based on a combination of tracks and transporters. It provides a set of attributes and methods to create the behavior of portal cranes. It can move several portals on to one crane runway. The multi-portal crane has a set of SimTalk attributes and methods (Table 6.8).

**Table 6.8** Multi-portal crane attributes and methods

Methods/Attributes	Description
<crane>.getPortals(tableFile list)	Writes references to the portals in a table and the table is automatically formatted
<portal>.endSequence	Sets the state of the portal crane to "idle" and should be called at the end of a load sequence
<portal>.moveHook(real position)	Moves the hook of the crane to the given position
<portal>.moveTo(real X, real Y, real Z)	Moves the portal to the specified position; if you pass -1 for Z, the hook is not moved
<portal>.moveToObject(object destination)	Moves the portal to the passed object
<portal>.moveToPosition(integer posX, integer posY)	Moves the portal to the passed position (pixel position in the network)
<portal>.state	State of the portal (waiting, driving, idle)
<portal>.hook	Returns a reference to the hook of the crane

The following example is used to explain the operation of the multi-portal crane object.

### Example: Multi portal crane

Three machines are loaded and unloaded through a portal crane. After repositioning on start, the parts are to be processed on M1, M2 or M3. After machining, the parts are moved to the station "Ende". Create a frame like in Fig. 6.83.

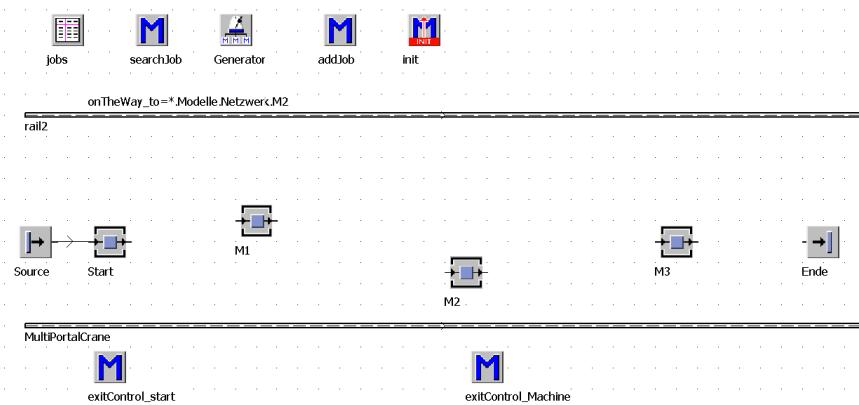


Fig. 6.83 Example frame

Double-click the multi-portal crane object to configure it. Ensure the following settings: length of the crane track: 40 meters; width of the crane track: ten meters; number of portals: one; portal length: three meters; portal width: ten meters; portal height: ten meters; overhang: zero meters; speed: 1.3 m/s; trolley lifting height: ten meters; trolley speed: 1 m/s; hook speed: 0.25 m/s. The source works with an interval of two minutes; M1 has a processing time of ten minutes, M2 of 12 minutes and M3 of seven minutes. Format the table jobs as in Fig. 6.84.

	object 1	object 2
string from		to
1		

Fig. 6.84 Table Jobs

The init method clears at the beginning of the simulation the content of the table jobs and sets the portal crane to "idle" for the first time.

```

is
do
  jobs.delete;
  multiPortalCrane.cont.endSequence;
end;

```

The method addJob takes the source object and the destination object as parameters and enters them into a new row in the table jobs:

```
(source:object,destination:object)
is
do
  jobs.writeRow(1,jobs.yDim+1,source,destination);
end;
```

### Load and unload using the PortalCrane

A transport with the PortalCrane takes place in several steps:

1. Wait until the crane state becomes "idle".
2. Move the empty portal to the machine.
3. Lower the crane hook.
4. Hang (move) the part on the hook.
5. Pull up the crane hook.
6. Move the crane to the next position.
7. Lower the hook with the part.
8. Move the part from the hook to the machine.
9. Pull up the hook.
10. By calling endSequence you release the portal for the next job.

Since more requests for the PortalCrane may arrive in the meantime, we will store the requests (jobs) for the crane in a table. The first request of a transport takes place at the exit of Start (exitControl\_start, front). The exit control waits for a free station and generates an order for the crane (addjob). The variable onTheWay\_to is used to prevent more than one job being created for one transport-destination.

```
is
  target:object;
do
  --search empty machine and create job
  waituntil (m1.empty and onTheWay_to /= m1) or
  (m2.empty and onTheWay_to /= m2) or
  (m3.empty and onTheWay_to /= m3)  prio 1;
  if (m1.empty and onTheWay_to /= m1) then
    target:=m1;
  elseif (m2.empty and onTheWay_to /= m2) then
    target:=m2;
  elseif (m3.empty and onTheWay_to /= m3) then
    target:=m3;
  end;
  addJob(?,target);
  onTheWay_to:=target;
end;
```

The machines M1, M2 and M3 have the same exit control (ExitControl\_Machine). The exit control creates a transfer order from the machine to the drain (Ende).

```
is
do
  addJob(?,ende);
end;
```

The control of the crane itself takes place in the method searchJob. The method searchJob is called every ten seconds by the generator. The method checks whether a job is to be executed, reads out the start and end, and controls the crane accordingly.

```
is
  start:object;
  target:object;
  portal:object;
do
  portal:=multiPortalCrane.cont;
  -- the crane has to be finished and a job must exit
  if portal.state="idle" and jobs.yDim>0 then
    --address the crane and delete the job
    start:=jobs[1,1];
    target:=jobs[2,1];
    jobs.cutRow(1);
    --go to start
    portal.moveToObject(start);
    waituntil portal.state="waiting" prio 1;
    --move hook down
    portal.moveHook(1.5);
    waituntil portal.state="waiting" prio 1;
    --hook part
    start.cont.move(portal.hook);
    --move hook up
    portal.moveHook(9);
    waituntil portal.state="waiting" prio 1;
    --go to target
    portal.moveToObject(target);
    waituntil portal.state="waiting" prio 1;
    --move hook down
    portal.moveHook(1.5);
    waituntil portal.state="waiting" prio 1;
    --unhook part
    portal.hook.cont.move(target);
    -- move hook up
    portal.moveHook(9);
    waituntil portal.state="waiting" prio 1;
    -- finish job
```

```

    portal.endSequence;
  end;
end;

```

### 6.10.2 *Simulation of a Forklift*

The simulation of forklifts must be set up as if the transport runs automatically. The transports could, e.g. be ordered from a location. The request has to contain all necessary information to allow the truck to fulfill its transport order. The requests are carried out successively by the forklift (or by a pool of forklifts). The forklift itself has to process a sequence of actions. It usually consists of:

- go to storage place and load a full container
- go to destination, unload a full container and load an empty container
- go to storage place and unload an empty container
- go to waiting position and wait for a new order

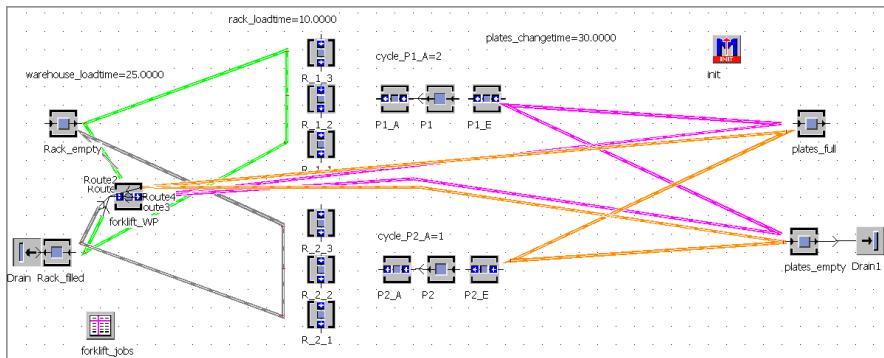
In the following example, we will program this sequence via a number of sensor controls.

#### **Example: Forklift**

You are intended to simulate the loading and unloading of presses. The forklift takes from the loading location of the press an empty pallet and loads the press with a plate stack. The forklift brings the empty plate palette to an empty containers warehouse. The finished product is stacked by a worker in racks. The forklift has to pick up the loaded racks and supply empty racks. Create a frame according to Fig. 6.85. To fulfill all the jobs, four different routes are necessary. They begin on the waiting point of the forklift (buffer) and end up there again. Insert the routes as follows:

- Route1: forklift \_Wp - Rack\_filled - R\_1\_3 - R\_1\_1 - Rack\_empty - forklift\_Wp
- Route2: forklift \_Wp - Rack\_filled - R\_2\_3 - R\_2\_1 - Rack\_empty - forklift\_Wp
- Route3: forklift\_Wp - plates\_full - P1\_E - plates\_empty - forklift\_Wp
- Route4: forklift\_Wp - plates\_full - P2\_E - plates\_empty - forklift\_Wp

Insert in the class library the following MUS: plate (entity), pallete (container capacity: 150), rack (container capacity: 20), forklift (transporter capacity: two; speed: 0.75 m/s). Connect all routes (exit) with forklift\_WP. Insert in the class track (class library) a method sensorControl (user-defined attribute type method). Insert on the routes, sensors (in the direction of the route) and assign to each sensor the method sensorControl as control. Insert on the routes, sensors to load the plates at place Plates\_full, one at the unloading place (e.g. P2\_E) and one at Plates\_empty. To change the rack, create a sensor at the station Rack\_empty, one sensor for each rack place (R\_1\_1 ...) and one sensor at place Rack\_filled. Enter in the presses (P1, P2) a processing time of 12 seconds and in the places for the exit of the presses a processing time of seven seconds.



**Fig. 6.85** Example frame

### Plates and Racks

The storage of plates and racks should not be part of the investigation. Therefore, it is sufficient to have a pallet with plates and an empty rack available if you need one. This can be easily realized using exit controls. Place an exit control (rear) in the station Plates\_full and Rack\_empty. In the exit control of the place Plates\_full, a pallet is created and on the pallet plates, until it is full:

```
is
  pallete:object;
do
  pallete:=.MUs.Pallete.create(?);
  while not pallete.full loop
    .MUs.Plate.create(pallete);
  end;
end;
```

On the place Rack\_empty, an empty rack is generated:

```
is
do
  .MUs.Rack.create(?);
end;
```

### Drive orders and forklift control

In this example, it is sufficient for a drive order to send the forklift to a specific route. Through the route (and the sensor programming), the path and the actions of the forklift are uniquely determined. Therefore, the table forklift\_jobs must contain only the name of the route in the first column. Add an exit control (front) in forklift\_WP. The truck has to wait until there is at least one entry in the table forklift\_jobs. Then, the control reads the route, removes the entry from the table forklift\_jobs and sets the forklift on the right route.

```

is
do
  waituntil forklift_jobs.yDim > 0 prio 1;
  --move to the specified route
  @.move(str_to_obj(forklift_jobs[1,1]));
  --delete job
  forklift_jobs.cutRow(1);
end;

```

### *Initializing*

The init method creates an empty rack on Rack\_empty and each empty rack on R\_1\_1 to R\_2\_3, creates a forklift on forklift\_WP and a full pallet on Plates\_full. To start the simulation, the init method creates the first two jobs for loading P1\_E and P2\_E (Route3 and Route4).

```

is
  pallete:object;
do
  -- create rack und pallets
  .MUS.Rack.create(rack_empty);
  .MUS.Rack.create(R_1_1);
  .MUS.Rack.create(R_1_2);
  .MUS.Rack.create(R_1_3);
  .MUS.Rack.create(R_2_1);
  .MUS.Rack.create(R_2_2);
  .MUS.Rack.create(R_2_3);
  pallete:=.MUS.Pallete.create(Plates_full);
  while not pallete.full loop
    .MUS.Plate.create(pallete);
  end;
  --insert forklift
  .MUS.forklift.create(forklift_WP);
  --create the first two jobs for the forklift
  forklift_jobs[1,1]:="Route3";
  forklift_jobs[1,2]:="Route4";
end;

```

The forklift should now be running two "empty rounds" and wait at the end on the station forklift\_WP for the next order.

### *Load the press with plates, unload empty pallets*

For the loading of pallets loaded with plates, the forklift must stop, move the pallet to the forklift and start driving again. To take into account the loading time, the method will be interrupted for the duration of the loading time (warehouse\_loadtime). At the unloading place of the press, the forklift must change the empty pallet with the full pallet. To do this, we can move the empty pallet to the second place of the forklift and take out the full one from the first

place. On the place Plates\_empty, the empty pallet is finally unloaded from the forklift. For the Route3 (press P1), this could look like:

```
(SensorID : integer; Rear : boolean)
is
do
  @.stopped:=true;
  if sensorID=1 then
    --load filled pallet
    plates_full.cont.move(@);
    wait(warehouse_loadtime);
  elseif sensorID=2 then
    if P1_E.occupied then
      --change empty to full pallet
      P1_E.cont.move(@);
      @.pe(1,1).cont.move(P1_E);
    else
      --only unload
      @.pe(1,1).cont.move(P1_E);
    end;
    wait(plates_changetime);
  elseif sensorID=3 then
    if @.occupied then
      --unload the forklift
      @.cont.move(plates_empty);
    end;
    wait(warehouse_loadtime);
  end;
  @.stopped:=false;
end;
```

*Load the plate on the press and the work of the press*

The loading and unloading of the press, you can realize well with exit controls in the pallet places and the press. On arrival of a full pallet, the first plate is moved by the exit control (front) of the place to the press. In the exit control (rear) of the press, a new plate is moved from the pallet to the press. When the pallet is empty, a new job is entered into the table forklift\_jobs. The exit control of the place P1\_E should look like this:

```
is
do
  --move the first plate to p1
  @.cont.move(P1);
end;
```

The exit control of the press P1 requires the following content:

```

is
do
  --get a new part
  if P1_E.cont.occupied then
    P1_E.cont.cont.move(P1);
    --if the pallet after moving the part is empty
    --then create a new job for the forklift
    if P1_E.cont.empty then
      forklift_jobs.writeRow(1, forklift_jobs.yDim+1,
                            "Route3");
    end;
  end;
end;

```

*Distribute the pressed parts to the racks*

After the pressing, the finished parts must be distributed to the racks. The racks are to be filled one by one. If a rack is full, a transport job is triggered. This is done in the exit control of the press exit places. For the place P1\_A, it should look like this (exit control front):

```

is
  rackplace:object;
do
  -- select rack using a cycle
  if cycle_P1_A=1 then
    rackplace:=R_1_1;
  elseif cycle_P1_A=2 then
    rackplace:=R_1_2;
  else
    rackplace:=R_1_3;
  end;
  waituntil rackplace.occupied prio 1;
  --wait for a change, if the rack is still full
  if rackplace.cont.full then
    waituntil rackplace.empty prio 1;
    waituntil rackplace.occupied prio 1;
  end;
  --move
  @.move(rackplace.cont);
  --if full, enter job into forklift_jobs and
  --increment cycle
  if rackplace.cont.full then
    forklift_jobs.writeRow(1, forklift_jobs.yDim+1,
                          "Route1");
    cycle_P1_A:=cycle_P1_A+1;
    if      cycle_P1_A=4 then

```

```

    cycle_P1_A:=1;
  end;
end;
end;

```

*Replace the full racks by empty racks*

To change the rack, the forklift must load an empty rack on his route, then replace the full rack against the empty one and unload the full rack. For the Press P1 and Route1, the sensor control should have the following content:

```

(SensorID : integer; Bug : boolean)
is
do
  @.stopped:=true;
  if SensorID=1 then
    --load empty rack
    rack_empty.cont.move(@);
    wait(warehouse_loadtime);
  elseif SensorID=2 then
    --change full rack >> empty rack
    if R_1_3.cont.full then
      R_1_3.cont.move(@);
      wait(rack_loadtime);
      @.pe(1,1).cont.move(R_1_3);
    end;
  elseif SensorID=3 and @.cont.empty then
    --change full rack >> empty rack
    if R_1_2.cont.full then
      R_1_2.cont.move(@);
      wait(rack_loadtime);
      @.pe(1,1).cont.move(R_1_2);
    end;
  elseif SensorID=4 and @.cont.empty then
    --change full rack >> empty rack
    if R_1_1.cont.full then
      R_1_1.cont.move(@);
      wait(rack_loadtime);
      @.pe(1,1).cont.move(R_1_1);
    end;
  elseif SensorID=5 then
    --load full to Rack_full
    @.cont.move(Rack_filled);
    wait(warehouse_loadtime);
  end;
  @.stopped:=false;
end;

```

The controls are very easy to customize for the second press.

# Chapter 7

## Simulation of Robots and Handling Equipment

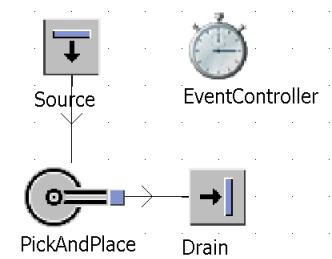
Using Plant Simulation, you can model robots and handling equipment in various levels of detail. For this purpose, Plant Simulation provides some blocks.

### 7.1 PickAndPlace

From Version 9 onward, Plant Simulation provides the object PickAndPlace. You can easily model robots with it, picking up parts at one position, and rotating and placing the parts at another position. Plant Simulation determines the necessary angles of rotation according to the position of the successors. Alternatively, you can enter the angles into a table.

#### Example: PickAndPlace 1

Create a frame as in Fig. 7.1.

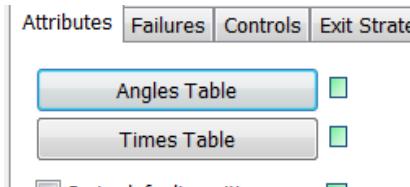


**Fig. 7.1** Example Frame

The source produces one part every minute. All other components use their basic settings. When you start the simulation, the PickAndPlace robot transports parts from the source to the drain.

### 7.1.1 Attributes of the PickAndPlace Object

If you connect the PickAndPlace robot with other objects using connectors, Plant Simulation generates a table of positions and associated objects. You can find the table in the tab Attributes. Click the button Angles Table (Fig. 7.2).



**Fig. 7.2** Angles Table

The position of  $0^\circ$  corresponds to the so-called three o'clock position. The angles are specified clockwise (Fig. 7.3).

	Name	Angle
1	Source	270.00
2	Drain	0.00

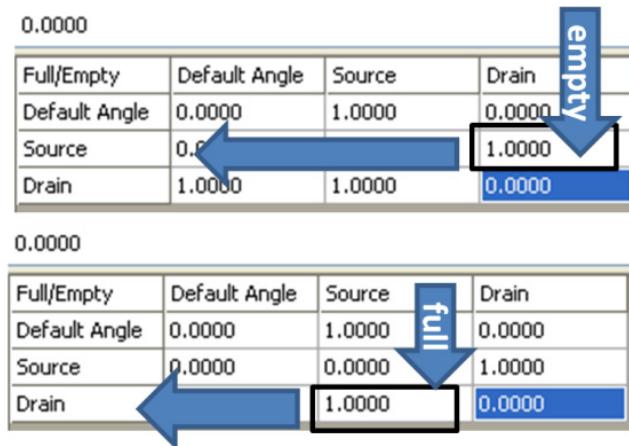
**Fig. 7.3** Angles Table

The Times Table controls the time consumption between the different rotation positions. Here, you define the duration of the movement from one position to another. The default is one second. To change the times, click the button Times Table on the tab Attributes (Fig. 7.4). The default angle designates a waiting position, which the PickAndPlace robot adopts when you select the respective option on the tab Attributes.

Full/Empty	Default Angle	Source	Drain
Default Angle	0.0000	1.0000	0.0000
Source	0.0000	0.0000	1.0000
Drain	0.0000	1.0000	0.0000

**Fig. 7.4** PickAndPlace Times Table

Enter the duration of the rotations as in Fig. 7.5.



Full/Empty	Default Angle	Source	Drain
Default Angle	0.0000	1.0000	0.0000
Source	0.0000	1.0000	1.0000
Drain	1.0000	1.0000	0.0000

Full/Empty	Default Angle	Source	Drain
Default Angle	0.0000	1.0000	0.0000
Source	0.0000	0.0000	1.0000
Drain	1.0000	1.0000	0.0000

Fig. 7.5 Entries in the Times Table

### Go to Default Position

In its basic setting, the PickAndPlace robot waits at the unloading position until a new part is available at the loading position. It then turns to the loading position and loads the part. If you select “Go to standard position,” the robot moves to this position after placing the part (Fig. 7.6).

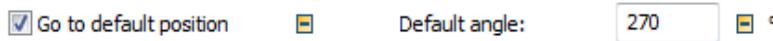


Fig. 7.6 Default Position

### Example: PickAndPlace2

A PickAndPlace robot is used to sort parts. Red, green, and blue parts arrive in mixed order. The robot distributes these according to the value of the attribute “col” of the parts. Create three parts (part1, part2, part3) in the class library. Assign a user-defined attribute (“col”; data type: string) to all three parts. Set the values to “red”, “green”, and “blue”. Color the parts accordingly. Create a frame according to Fig. 7.7.

Settings: The source randomly creates part1, part2, and part3 with a share of 33 per cent, each within intervals of two seconds. Connect the PickAndPlace object with Line\_red first, then with Line\_green, and lastly with Line\_blue. Define the distribution of parts by color using the exit strategy of the PickAndPlace robot. Select the option MU Attribute on the tab Exit Strategy. Then click Apply. The dialog shows additional dialog items (Fig. 7.8).

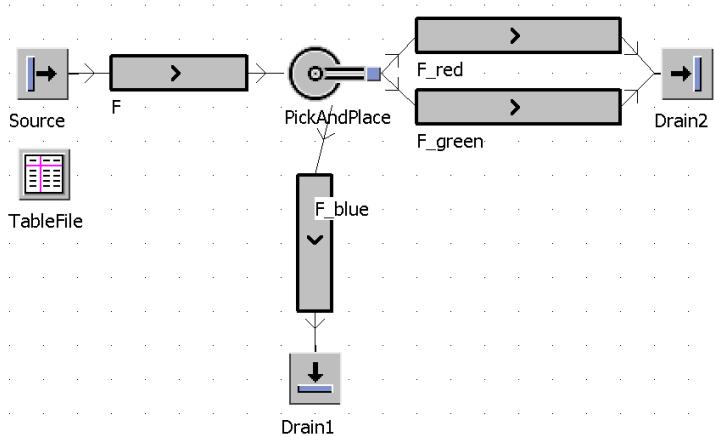


Fig. 7.7 Example Frame

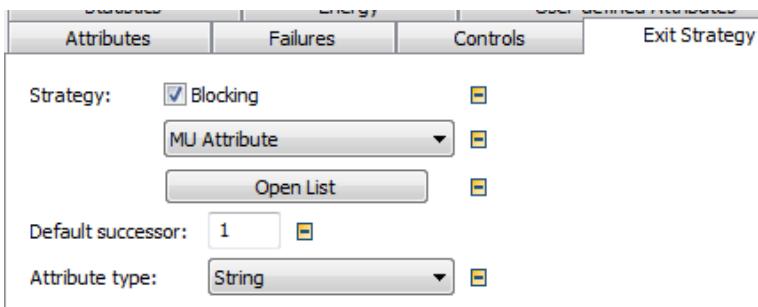


Fig. 7.8 PickAndPlace Exit Strategy

Select the Attribute type as String (the data type of the user-defined attribute col). Click Open List. Enter the attributes, the values and the successors to be transferred here (Fig. 7.9).

	Attribute	Value	Successor
1	col	blue	1
2		red	2
3		green	3

Fig. 7.9 Attribute Table

Finally, the robot should place the parts assorted by color to the lines.

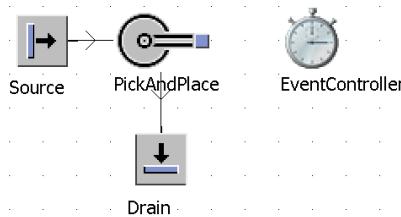
### 7.1.2 Blocking Angle

Often, a robot cannot rotate a full  $360^\circ$ . This may be due to structural constraints or the construction of the robot. Many robots have hose packages that are fixedly

connected with aggregates. If such a robot rotates once around himself, the robot or the environment of the robot would be damaged. Plant Simulation uses for calculation of the rotation the smallest distance between two angles. When you define a blocking angle at that location, Plant Simulation selects the longer route. The relevant time values must be entered in the Times Table by hand.

### Example: PickAndPlace Blocking Angle

Create a frame like in Fig. 7.10.



**Fig. 7.10** Example Frame

Check the Angles table of the PickAndPlace. The source is located at the angular position of  $180^\circ$  and the drain at  $90^\circ$ . "Go to default position" is unchecked. The source produces one part each minute. When you start the simulation, Plant Simulation selects the shortest path between the two positions. The robot in the example cannot rotate about the angular position  $135^\circ$  (in the example, "left down"). Set in the tab Attributes a blocking angle of  $135^\circ$  (Fig. 7.11).



**Fig. 7.11** Blocking Angle

The robot now uses the longer path to transport the MUs. Note that only the animation is adjusted. Consider the longer time required in the Times Table from source to drain and vice versa.

### 7.1.3 Time Factor

Many robots have different rotational speeds at different workpiece weights. Hence, the complete Times Table need not be re-filled for all changes in the workpiece weight or other factors; you can set a factor by which all times are multiplied here.

## 7.2 Simulation of Robots

For robot simulations, various approaches exist in Plant Simulation. Depending on the required level of detail, robots can be modeled using the track and transporter or with the PickAndPlace object of Plant Simulation. The creation of the controls can be carried out in different ways. The following will present various practical examples of these options. The examples are outlines of a nearly unmanageable variety of applications of industrial robots. The focus is on the use of robots in material flow. Typical application fields are:

- Pick and place
- Assembly
- Load and unload of machines

### 7.2.1 *Exit Strategy Cyclic Sequence*

With the exit strategy of cyclic sequence, you can model cases wherein a robot is used to divide one flow of material in several flows—e.g. a robot loaded cyclically a range of conveyor lines. This task can be solved without SimTalk. Select the tab Exit strategy in the field strategy Cyclic sequence and enter in the list the sequence of successors that the robot has to supply in sequence. After the end of a sequence of the robot, the sequence starts over again.

#### Example: PickAndPlace3

A robot has to distribute parts from a conveyor line uniformly to three conveyor lines. Every 10 seconds, a part reaches the robot (setting source interval). Create a frame according to Fig. 7.12.

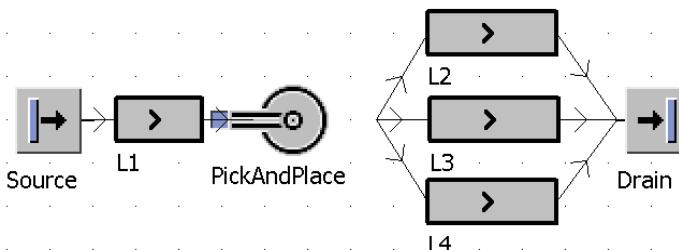
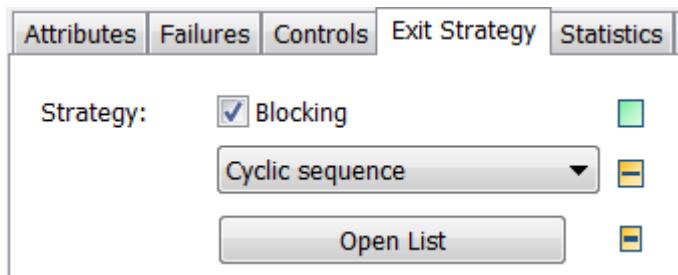


Fig. 7.12 Example Frame

Create settings according to Fig. 7.13 in the tab Exit Strategy of the PickAndPlace.



**Fig. 7.13** Exit Strategy Cyclic Sequence

Enter in the list a sequence such as 1, 2, 3 or 2, 3, 1. To ensure that the robot swings back again after each unloading, choose in the tab Attributes the option "Go to default position" and enter a default angle of 180°.

### 7.2.2 *Load and Unload of Machines (Single Gripper)*

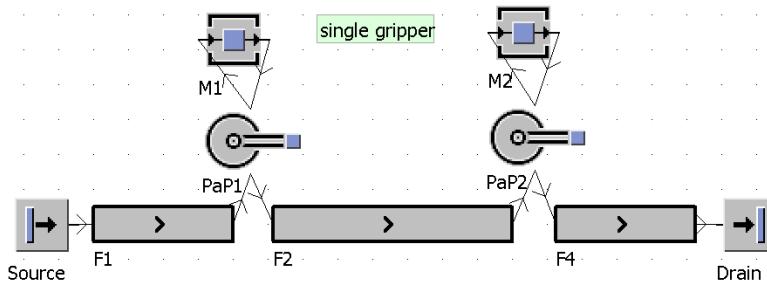
Loading and unloading with one gripper can be simulated easily (the robot can load only one part; there is no interim storage place). The sequence that the robot has the work through is as follows:

1. The robot loads the MU and turns to the machine.
2. The robot loads the MU into the machine and is waiting in front of the machine until the process is complete.
3. The robot picks the MU from the machine, swings to the unloading position and places the MU there.
4. Thereafter, the robot turns to the loading position and (is waiting) loads a new MU; thus, the sequence begins again.

In relation to Plant Simulation, consider the following facts: The robot responds to each request. When the robot has to process a more complex sequence, you need to control the requests to the robot. In the sequence above, this means that the new MU may trigger the request for transportation only when the machine has completed its process and is empty.

#### **Example: Load Machine with Robot (Single Gripper)**

Two machines in a line are loaded with robots. The robot can carry only one part at a time. The processing time of the machine is 35 seconds, with a new part arriving every 40 seconds. Create a frame as in Fig. 7.14.



**Fig. 7.14** Example Frame

If you enter availabilities in the machine or vary the processing times, parts may accumulate prior to the machining. In this case, prevent the calling of the robot before the machine and the robot are empty—e.g. exit control F1:

```
is
do
  waituntil M1.empty and PaP1.empty prio 1;
  @.move;
end;
```

### 7.2.3 Load and Unload of Machines (Double Gripper)

In practice, loading with single grippers often leads to long loading times. The machine must wait until after the robot has unloaded the finished MU from the machine, moved to the unload position, placed the finished part there, turned to the loading place of the next MU, where he has loaded a new MU, turned back to the machine and placed this MU into the machine. The time here can be considerably reduced by using a double gripper. One problem of the PickAndPlace object is that although it can load several parts (starting from Version 11), transportation begins only when it is fully loaded or empty. With a little trick, you can solve the problem without a lot of programming. You need to change the loading sequence. After the first loading of the empty machine (init method), the robot picks another MU (although the machine is busy). For the unloading, add after the machine a buffer in the model. The robot waits with the part in front of the full machine, until it is finished. The machine then moves the part into the buffer (connector). The PickAndPlace robot loads the MU into the machine, takes the finished MU from the buffer and turns to the finished MU position. This comes close to the behavior of a robot with a double gripper.

#### Example: Load and Unload Machines with Robot (Double Gripper)

Create a frame according to Fig. 7.15.

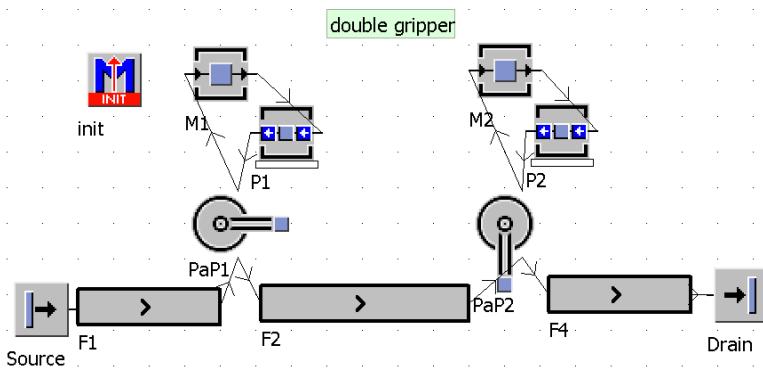


Fig. 7.15 Example Frame

The processing time of the machine is 35 seconds and every 40 seconds a new part arrives. P1 and P2 each have a capacity of one and no processing time. To prevent the robot cycle being messed up, it is necessary to initialize the simulation with some parts. We need one part on each machine and one on the feeding conveyors, so that the initial state in the cycle is established (robot waits with the part in front of an already loaded machine). To do this, insert a method in the frame and name it "init." The init method should have the following content:

```

is
do
  .MUS.part.create(M1);
  .MUS.part.create(F1);
  .MUS.part.create(M2);
  .MUS.part.create(F2);
end;

```

Add exit controls in the conveyor lines F1 and F2. A part is loaded onto the PickAndPlace when the robot is empty and the auxiliary buffer on the machine is empty. The machine can be occupied while the robot is loading the MUs (cycle time parallel loading). The exit control of the conveyor line F1 would then look like this:

```

is
do
  waituntil P1.empty and PaP1.empty prio 1;
  @.move;
end;

```

### 7.2.4 *PickAndPlace Loads Containers*

Create a frame as in Fig. 7.16.

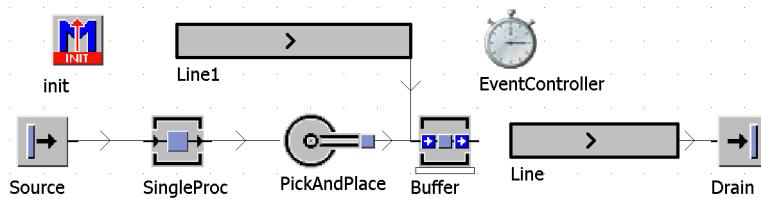


Fig. 7.16 Example Frame

Set a capacity of eight in the container in the class library. The SingleProc has a processing time of one second. The PickAndPlace has a capacity of four and a blocking angle of 270 degrees. The source creates containers. The exit control of the source fills the containers with entities:

```
is
do
  while not @.full loop
    .MUs.Entity.create(@);
  end;
end;
```

In the exit control of the SingleProc, the first four MUs are moved to the PickAndPlace. In order to move the second four MUs to the PickAndPlace, the container is rearranged to the SingleProc again to re-trigger the exit control. If the container is empty, it will move to Line1. Exit control (front) of SingleProc:

```
is
do
  --move 4 MUs to the PickAndPlace
  @.cont.move(PickAndPlace);
  @.cont.move(PickAndPlace);
  @.cont.move(PickAndPlace);
  @.cont.move(PickAndPlace);
  if @.empty then
    @.move(line1);
  else
    --for the second 4 unloading
    --re-trigger the exit control
    @.move(?);
  end;
end;
```

First, the PickAndPlace robot has to wait until a container is present on Buffer1. It can then move the loaded MUs to the container. If the container is full, it is rearranged to Line. Exit control of PickAndPlace:

```

is
do
    --wait for container
    waituntil buffer.occupied prio 1;
    --move MUs to container
    ?.cont.move(buffer.cont);
    ?.cont.move(buffer.cont);
    ?.cont.move(buffer.cont);
    ?.cont.move(buffer.cont);
    --move full containers to line
    if buffer.cont.full then
        buffer.cont.move(line);
    end;
end;

```

### 7.2.5 Assembly with Robots

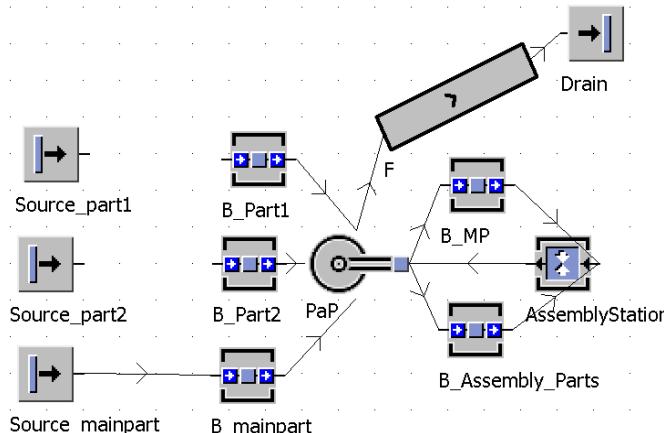
A typical scenario might look like this: A robot takes one part at one position (carrier, main part) and puts it into an assembly (clamping) station. The robot then takes parts from containers at different positions and places them on the main part in the assembly station. Following assembly, the robot removes the assembled main part from the assembly station and places it on a conveyor line. There are numerous similar scenarios. For such tasks, the PickAndPlace object can also be used. While working with the assembly station, the following is important: The supply of the main part and the add-on parts must be carried out through different predecessors if you want to use the assembly mode "Attach MUs." Therefore, you need separate places for the supply of parts to the assembly station. Transport to the assembly station should take place according to the assembly sequence; this means, first the main part and then the add-on parts should be moved. The order of the transports of the robot can be solved easily via a cyclic sequence:

1. Main part to place main part
2. All add-on parts to place add-on parts
3. The assembly to the exit of the process

During the assembly process, all the other arriving parts must be blocked in order to prevent the triggering of false requests of the robot (in the cycle at the position in which the robot is next scheduled to transport the finished part to the exit). For this task, very little SimTalk programming is necessary.

### Example: Assembly with Robot

A robot loads two parts on a main part. The parts are welded together. The finished part is placed by the robot on a conveyor belt. The welding operation takes 45 seconds. In 65 seconds, a main part reaches the assembly cell. Create a frame as in Fig. 7.17.



**Fig. 7.17** Example Frame

All buffers have a capacity of one and a processing time of zero seconds. The assembly station has a processing time of 45 seconds; the assembly mode is "Attach MUs"; assembly table: predecessor (two parts from B\_Assembly\_parts). Connect the PickAndPlace station as follows with the blocks (exits):

1. PaP – B\_MP (Successor 1)
2. PaP – B\_Assembly\_Parts (Successor 2)
3. PaP – F (Successor 3)

The successor to the assembly station is again the PickAndPlace robot. The numbers of successors can be displayed in the frame via View—Options—Show Successors. You can correct the angular positions, which determine the PickAndPlace object, at any time in the Angle Table (tab Attributes). Set the angular position of the stations B\_MP, B\_Assembly\_Parts and AssemblyStation to 360°. The PickAndPlace station must transport the MUs in a sequence: B\_MP (the main part first), B\_Assembly\_Parts, B\_Assembly\_Parts (then two add-on parts), F (the final part of the assembly station to the conveyor line). You can do this easily with the exit strategy "Cyclic sequence". The sequence is 1, 2, 2, 3. Just ensure that no other MU requests the PickAndPlace robot before completing the cycle. The procedure is as follows: As you can see, only one source is connected with the buffer. Insert an exit control in the source\_mainpart (front). In this control, the source waits with the repositioning of the main part until B\_MP, P\_Assembly\_Parts, AssemblyStation and PaP are empty (the cycle is complete). Then the part is moved:

```

is
do
    waituntil B_MP.empty and B_assembly_Parts.empty
        and PaP.empty and assemblyStation.empty prio 1;
    @.move;
end;

```

The add-on parts Part1 and Part2 will be moved to the buffers when the main part has left B\_mainpart. Set in the sources an interval of zero (blocking) for every time that parts are available. Exit control B\_mainpart (rear):

```

is
do
    source_part1.cont.move(b_part1);
    source_part2.cont.move(b_part2);
end;

```

When the parts are provided by a previous process, it may be necessary under certain circumstances to wait before moving the parts. At the end, you can position the individual elements of the assembly station one on top of the other (Fig. 7.18).

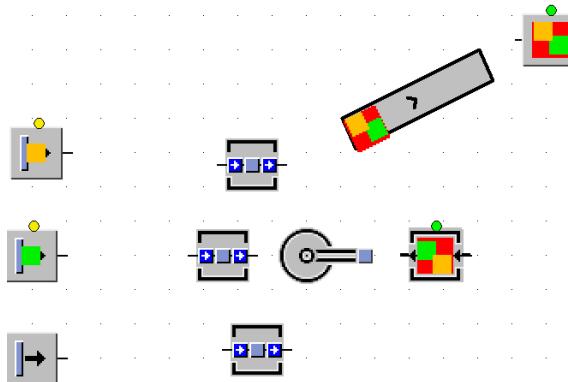


Fig. 7.18 Assembly Cell

### 7.2.6 The Target Control of the PickAndPlace Object

You can use the target control of the PickAndPlace object to set the destination of the robot dynamically. The target control is called when the robot has loaded an MU. You can then specify the destination of the transport on the basis of MU attributes. With the help of the target control, for example, you can easily model assembly processes.

#### Example: Target Control

Create a frame as in Fig. 7.19.

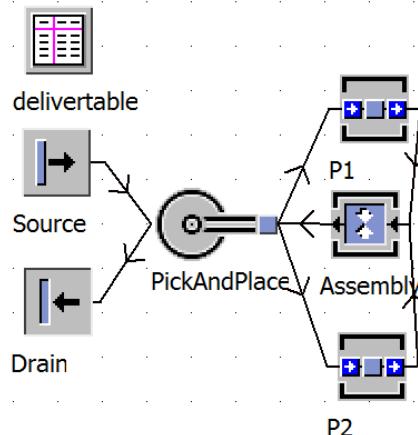


Fig. 7.19 Example Frame

You should model an assembly process. The robot places parts in P1 and P2 sequentially (parallel to the cycle time of the assembly station), takes away the fully assembled part from the assembly station and transports it to the drain. Insert three entities (P1, P2, P3) in the class library. Create the following settings in the frame: The source produces sequentially one entity P1 and P2 at intervals of 10 seconds (MU-selection sequence: cyclical; table: deliverable). The buffers P1 and P2 have a capacity of four and no processing time. The assembly station has the following settings: processing time: 18 seconds; assembly table: predecessors (one part of P2); main MU from predecessor: 1; assembly mode: delete MUs; exiting MU: new MU (P3). To map the assembly process precisely, the robot has to transport part P1 to buffer P1, P2 to buffer P2 and P3 (after assembly) to the drain. Assign a method for the target control of the PickAndPlace (Tab controls, key F4). The assignment of the target to the robot takes place by calling the method `<path>.setDestination (<object>, <boolean>)`. The first parameter determines the destination of the robot, the second parameter if the robot has to remain at this position after unloading. In this example, the method would appear as follows:

```

is
do
  --?.setDestination(?.succ, false /* true
  --if robot should wait at target */;
  if @.name=="P1" then
    ?.setDestination(P1, false);
  elseif @.name=="P2" then
    ?.setDestination(P2, false);
  elseif @.name=="P3" then
    ?.setDestination(Drain, false);
  else
    debug;
  end;
end;

```

### 7.2.7 Consider Custom Transport and Processing Times

You can set the duration from one position to another in the Times Table (tab Attributes—button Times Table). However, if you need to serve several stations and move to a default position, then the entry becomes confusing quickly. It is better to create a small interface in which you can enter your cycle times, as determined by the planning. To create the corresponding function, you need the methods from Table 7.1 of the PickAndPlace object:

**Table 7.1** SimTalk Methods PickAndPlace Times Table

Method	Description
<path>.getTimesTable(<tableFile>)	Copies the contents of the times table into the passed table
<path>.setTimesTable(<tableFile>)	Sets the contents of the time table of the PickAndPlace object to the contents of the passed table

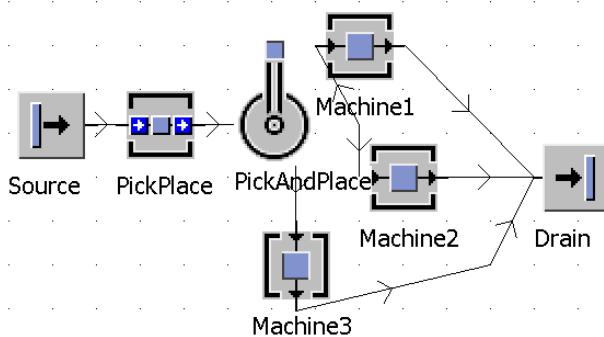
The approach is simple. First, copy the Table into a temporary table. The method `getTimesTable` formats the temporary table and transmits all the data. The old times must be deleted. You then transfer all the values from the input table to the temporary times table. At the end, copy the values from the temporary table into the times table of the PickAndPlace object. The input table can then be set up according to your requirements.

#### Example: PickAndPlace, User-defined Times Table

A PickAndPlace robot feeds three machines. For all three machines, it has different processing times. After each loading, it returns to a default position (180°). A planning process found times according to the following table.

from	to	Moving time	Process time
PickPlace	Machine1	3 sec	10 sec
PickPlace	Machine2	4 sec	15 sec
PickPlace	Machine3	5 sec	9 sec
Machine1	Default	2 sec	
Machine2	Default	3 sec	
Machine3	Default	4 sec	
Default	PickPlace	2 sec	

Duplicate the PickAndPlace class in the class library. Use the duplicate in the following example. Create a frame according to Fig. 7.20.



**Fig. 7.20** Example Frame

Ensure the following settings:

Block	Attribute	Value
Source	Interval	20 sec.
PickPlace	Capacity	1
	ProcTime	0 sec.
PickAndPlace	Go to default position	
	Default angle	180°
Machine1 to 3	ProcTime	30 sec.

The programming should be done in the class (duplicate of the class PickAndPlace). In this way, you can use the functionality in other simulation models (save the class as an object and import it into another file again). First, create three user-defined attributes in the class:

- init (data type method)
- times\_table (type table)
- temp\_table (type table)

Format the times\_table according to Fig. 7.21 and enter the data.

	string 1	string 2	real 3	real 4
string	from	to	Movingtime	Processtime
1	PickPlace	Machine1	3.00	10.00
2	PickPlace	Machine2	4.00	15.00
3	PickPlace	Machine3	5.00	9.00
4	Machine1	Standardwinkel	2.00	
5	Machine2	Standardwinkel	3.00	
6	Machine3	Standardwinkel	4.00	
7	Standardwinkel	PickPlace	20.00	
8				

**Fig. 7.21** Times Table

Within the init method, the data in the Times Table of the PickAndPlace object will be rewritten. This is done in four steps:

1. The current times table from the PickAndPlace object is copied to the temp\_table. If you have correctly connected all objects with the PickAndPlace object then the times table will contain all necessary station names as row and column indices.
2. All values are deleted from the temp\_table.
3. The temp\_table is re-filled with the values from the internal times\_table thereby, the column and row indices are used.
4. The Times Table of the PickAndPlace object is re-filled with the contents of temp\_table.

The method init has the following content:

```

is
i:integer;
from_:string;
to_:string;
do
  self.~.getTimesTable(self.~.temp_table);
  --delete all values
  self.~.temp_table.delete;
  -- fill new
  for i := 1 to self.~.times_table.yDim loop
    from_:=self.~.times_table["from",i];
    to_:=self.~.times_table["to",i];
    self.~.temp_table[from_,to_]:= 
      num_to_time(self.~.times_table["Movingtime",i] +
      self.~.times_table["Processtime",i]);
  next;
  -- write back the table
  self.~.setTimesTable(self.~.temp_table);
end;

```

The init method is called when the simulation is started by the event manager. The data should then be entered in the PickAndPlace station as shown in Fig. 7.22.

Voll/leer	Standardwinke	PickPlace	Machine1	Machine2	Machine3
Standardwinke			2.0000	3.0000	4.0000
PickPlace	20.0000				
Machine1		13.0000			
Machine2		19.0000			
Machine3		14.0000			

**Fig. 7.22** Times Table

### 7.2.8 *Advantages and Limitations of the PickAndPlace Object*

With the help of the PickAndPlace object you can model robots relatively easily. The object offers statistical analysis and is easily configured. More complex handling sequences with more than one input can only be realized with many tricks. Here, the modeling of logic quickly becomes confusing and needs to be distributed to different objects. More generally, you can apply the PickAndPlace robot of Plant Simulation with no problem if there is only one entrance. The distribution to several successors can be easily realized with sequences or user-defined attributes.

## 7.3 Model Handling Robots Using Transporter and Track

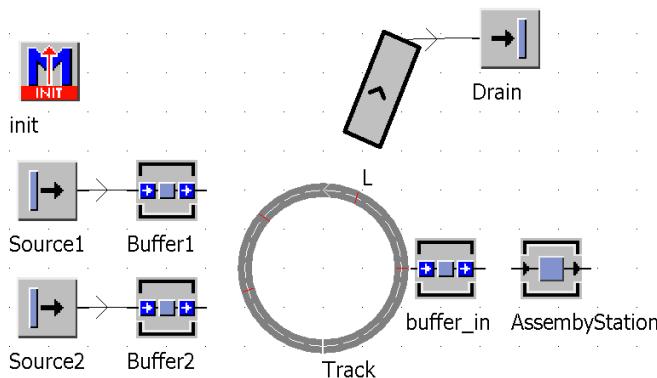
Robots can be modeled with track and transporter. To do this, we turn the transporter or let the transporter drive on a straight path back and forth.

### 7.3.1 *Basic Model and General Control*

The basic behavior of a handling robot comprises the loading of a part and the transfer of the part to a predetermined position. In doing so, the robot turns and possibly performs other movements in its axes. At the target location, the robot places the part and turns eventually to a waiting position. This basic behavior can be easily simulated with a transporter and a track. Plant Simulation rotates the icon of the transporter while driving in a curve so that the front of the transporter points in the direction of movement. If the transporter is on a circular path, then the symbol of the transporter rotates according to the direction of movement. If the icon is drawn correctly and the reference point is chosen correctly, then the icon turns around itself. The 2D animation then corresponds roughly to the behavior of a rotating handling robot.

#### **Example: Robot with Transporter Part 1**

You should develop a handling robot. Development takes place using the example of a simple scenario: The handling robot has to load two parts from different positions and place them in a device. After completion of the assembly process in the device, the robot removes the part and transports it to a conveyor. Then the cycle starts again. Create a frame as in Fig. 7.23.



**Fig. 7.23** Example Frame

Create the following settings:

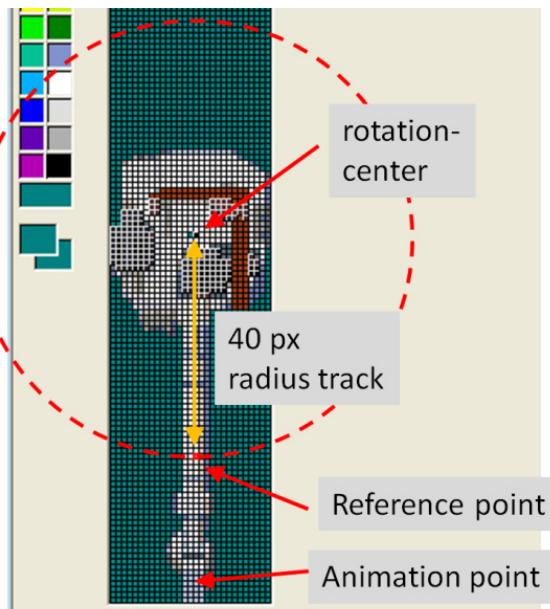
Block	Attribute	Value
Source1	Interval	30 sec.
	MU	Container
Source2	Interval	30 sec.
	MU	Entity
Buffer1	Capacity	1
Buffer2	ProcTime	0 sec.
AssemblyStation	ProcTime	20 sec.

The circular path can be created as follows:

1. Click on the Track in the Toolbox
  2. Press the key combination **Ctrl + Shift** and click in the frame.
  3. The “Edit Parameters of Curve” dialog is open; change the Center angle to  $360^\circ$ .
  4. Click on the top right from the previous position in the frame (while pressing **Ctrl + Shift**). If the track has been created, click the right mouse button and exit the function by pressing **Esc**.
- Note: The track must run counter-clockwise; otherwise, the animation will not work correctly.

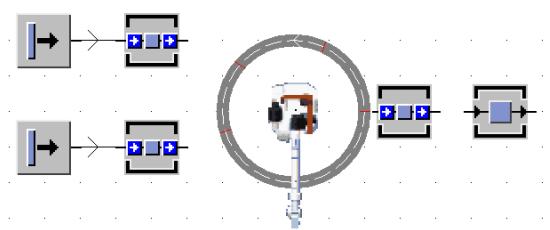
Drag a connection from the end of the track to the entrance (so that the transporter can move in a circle). You may require several attempts. On the second try, the key combination and two mouse-clicks usually suffice. The robot is an instance of the class Transporter. Duplicate the transporter in the class library and rename the duplicate as “robot”. The icon of the vehicle must be changed. In the Plant Simulation clip art library (Icon editor—Tools—Clip art library), you can find an icon for a robot in the folder Misc. Rotate the symbol by  $90^\circ$  (e.g. in Paint), so that

the gripper is facing down. Then, insert the picture into the icon editor. To ensure that the robot rotates correctly, change the icon size. So that the robot can rotate around its center, the icon must be 132 px high (and 29 px wide). The rotation center is approximately at the position (29, 66). Copy the symbol of the robot, so that the gripper is located at the bottom of the symbol. The track has a radius of 40 px. Place the reference point of the icon at the position (16, 106) (see Fig. 7.24). Specify the animation point on the position of the gripper. Make this setting for the icons operational, failed, paused and waiting (it is best when you create the setting for one icon, and copy the icon and paste it into the other icons).



**Fig. 7.24** Robot Icon

If you pull the robot on the track, then the robot should be in a position like in Fig. 7.25.



**Fig. 7.25** Handling Robot

Note: The transporter has a length of 1.5 meters by default. This has an impact on the position of the object on the track. In order to achieve accurate positioning (also with respect to the sensors and different directions), it is useful to set the length of the transporter to a very small value (e.g. 1 mm).

For the control, the transporter requires a user-defined attribute TargetSensorID of type integer. Add this attribute in the class library in the robot. As a next step, create a method for the sensor control in the track. To do this, open the track, select the tab User-defined attributes and add an attribute named sensorControl (type method). Set at all positions of the robot sensors. You can use the context menu of the track— - Create Sensors; start at buffer\_in and follow the direction of the track. Assign to each sensor the method sensorControl. The sensor assignment should look something like this:

Sensor number	Position (meters)	Near station
1	3.14	Buffer_in
2	5.4	L (line)
3	8	Buffer1
4	10	Buffer2

The robot should be placed on the track via an init method. The first target sensor of the robot should be 3. Insert a method in the frame and rename it as init. The init method should contain the following programming:

```
is
  robot:object;
do
  --create robot
  robot:=.MUS.Robot.create(track);
  --set the first target sensor
  robot.targetSensorID:=3;
end;
```

There are several ways to program the sensor control.

1. “Quick and Dirty”: There is only one robot in the simulation; a further use of the robot for other models is not intended. There is no parameterization. The creation of control is relatively easy (and takes very little time), but all changes must be made later in the method.
2. Partially Parameterized: Object references will be managed outside of the method (e.g. via a sensor list). The method still contains the control logic.
3. Fully Parameterized: Object references and logic are stored outside of the method (e.g. in tables). The development takes a little longer and debugging is difficult, but you can use fully parameterized objects in many simulations without changing the programming.

*SensorControl Variant “Quick and Dirty”*

We will develop the sensor control step by step. After each step, run the simulation to test the behavior of the robot. In this way, the controller can be debugged visually.

Basic model of the control: The robot is being addressed (attribute TargetSensorID). When the robot has reached its target position, it stops and performs various actions. After the action, the robot will be re-addressed, the direction will be chosen and the robot will be restarted. Therefore, the control consists of the following basic structure:

```
(SensorID : integer; isFront:boolean)
is
do
  if @.targetSensorID = SensorID then
    -- arrived at target
    @.stopped:=true;
    --different actions depends aon the
    --position
    if SensorID= 1 then
      -- do something
    end;
  end;
end;
```

The robot should perform the following actions: The robot moves empty to Sensor 3 and waits there until a pallette on Buffer1 is available. The robot loads the pallette and drives backward to Sensor 1, where it unloads the pallette (on buffer\_in) and moves back (forward) to Position 4. Here, it loads a part and turns back to Position 1, where it loads the part on the pallette (buffer\_in.cont) and moves the pallette to the assembly station. The robot waits until the assembly process is complete and loads the pallette from the assembly station and rotates to Sensor 2 (forward). On Sensor 2, the robot loads the part on the conveyor line and turns to Sensor 3. Then the cycle begins again. Similar tasks can be developed well in a stepwise manner (along the cycle). In the example above, start at SensorID 3. The following snippets are excerpts from the sensor control and are all located in the “if sensorID=...” query. The control of the SensorID 3 (Buffer1):

```
--excerpt from sensorControl

elseif SensorID=3 then
  -- load a pallet
  waituntil buffer1.occupied prio 1;
  buffer1.cont.move(@);
  --backwards to SensorID 1
  @.backwards:=true;
  @.targetSensorID:=1;
  @.stopped:=false;
```

At Sensor 1, we have a special situation. The pallet is loaded directly to the station buffer\_in, the part to buffer\_in.cont. When unloading, we must detect if the station is empty or occupied. In the first case, it is empty. In the next, you need at SensorID 1 the following programming:

```
--excerpt from sensorControl
if SensorID= 1 then
  -- if buffer_in empty then only unload and
  --drive to sensorID 4
  if buffer_in.empty then
    --unload
    @.cont.move(buffer_in);
    --forward to Sensor 4
    @.targetSensorID:=4;
    @.backwards:=false;
    @.stopped:=false;
  else
    --- load part on pallet
    debug; --remember
  end;
```

At Sensor 4, you load only the part of Buffer2 and then go to ID 1. This corresponds almost completely to the control of the SensorID = 3; you can copy anything and must change only the references. Control SensorID = 4:

```
--excerpt from sensorControl
elseif SensorID=4 then
  -- load one part
  waituntil buffer2.occupied prio 1;
  buffer2.cont.move(@);
  --backwards to SensorID 1
  @.backwards:=true;
  @.targetSensorID:=1;
  @.stopped:=false;
```

We load the part on the pallete on to Sensor 1 and move the pallete to the assembly station. You can use the attribute resWorking to wait for the completion of the assembly. Now, expand the sensorControl at SensorID = 1 as follows (else branch):

```
--excerpt from sensorControl
if SensorID= 1 then
  -- if pbuffer_in empty then only unload
  --and drive to sensorID 4
  if buffer_in.empty then
    --unload
    @.cont.move(buffer_in);
    --forward to Sensor 4
    @.targetSensorID:=4;
```

```

@.backwards:=false;
@.stopped:=false;
else
  --- load part on pallet
  @.cont.move(buffer_in.cont);
  -- move pallet to assembly station
  buffer_in.cont.move(assemblyStation);
  --wait, until assembly is finished
  waituntil assemblyStation.resWorking=false
    prio 1;
  -- unload assembly station
  assemblyStation.cont.move(@);
  -- forward to sensor 2
  @.targetSensorID:=2;
  @.backwards:=false;
  @.stopped:=false;
end;

```

On sensorID = 2 the robot unloads only onto the conveyor line and the cycle starts again from the beginning on SensorID = 3:

```

--excerpt from sensorControl
elseif SensorID=2 then
  -- unload
  @.cont.move(L);
  --forward to sensorID 3
  @.backwards:=false;
  @.targetSensorID:=3;
  @.stopped:=false;

```

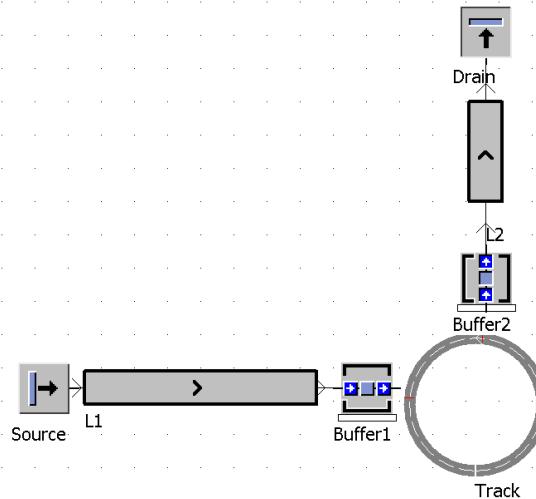
If you have to create only one robot, the procedure shown is sufficient. A big advantage is the step-by-step development and the "visual" debugging (you are able to observe if the robot does not do what it is supposed to do).

### 7.3.2 *Partial Parameterized Control Development*

In most cases, there is a unique mapping of a SensorID to a particular object. This assignment can be mapped easily into a table. Most of the tasks of the robot are similar at various locations: Load parts when the robot is empty and unload parts when the robot is occupied. The direction that the robot needs to follow on the track can be easily identified on the basis of the sensor positions. For simple handling tasks, it is possible to simplify the programming considerably in this way (one program for all sensors). Only the logic (the cycle of the robot) must be explicitly programmed for the individual sensor positions. Sensor lists are best defined in the robot track. In this way, you can create templates for specific types or classes of robots. You can also insert the init method to generate the robot in the robot track.

### Example: Robot with Transporter Part 2

You must simulate a simple transfer operation. As the simulation comprises many similar tasks that need to be simulated create a reusable object. Duplicate the track in the class library. Create a frame according to Fig. 7.26 and use for this the duplicate of the track.



**Fig. 7.26** Example Frame

Set the source to an interval of 30 seconds; Buffer1 and Buffer2 each have one place and no processing time. The track has a radius of two meters. Create the robot similar to the instructions from the previous example. The robot should be generated by an internal init method of the track. For this, insert in the track (duplicate in the class library) a custom attribute init, of the type method. For initialization, we need a reference to the robot class in the class library. So that the init method works with different robot classes without changes in the programming, you need a reference to the robot class as an attribute in the robot track (attribute: robot; type: object). The access to attributes of the same object from an internal method attribute is carried out by means of: `self.~.attributeName` ("self" points to the method itself and "self.~" to the location of the method—in our case, to the track). The init method should have the following content to generate the robot on the track:

```
is
do
  self.~.robot.create(self.~);
end;
```

You can now enter different robot addresses in the custom attribute `robot` and the method generates different robots, without changes in the programming. Insert in the track a user-defined method attribute (`sensorControl`) and generate the appropriate sensors (Fig. 7.27).

ID	Position	Front	Rear	L.	Path
1	6.07m	x			self.sensorControl
2	9.2m	x			self.sensorControl

**Fig. 7.27 Sensors**

Assign SensorID 1 to Buffer2 and SensorID 2 to Buffer1. This works best with a tableFile (in theory, also with a list with the sensor position as line number, but here you can get in trouble if you have one sensorID without an object, e.g. as a stand-by position, because you cannot leave a line blank). Insert in the track a user-defined attribute (type table). Name it SensorTable. Set the data type of the first column of the table to object. Enter in the first row Buffer2 and in the second row Buffer1. At the end, the table should look like Fig. 7.28.

	object	strin
1	object 1	2
2	root.Buffer2	
	root.Buffer1	

**Fig. 7.28 SensorTable**

You can now program the control of the robot in a simpler way: When the robot reaches its target sensor, it stops. If it is empty, it waits until the station from the sensorTable with the corresponding index is occupied. Then, it loads the part and turns to the respective other sensor (and changes the direction if necessary). If it is occupied, it waits until the assigned place is empty, and then moves the part to the place. SensorControl:

```
(SensorID : integer)
is
  station:object;
do
  station:=self.~.sensorTable[1,sensorID];
  if sensorID = @.targetSensorID then
    --arrived at target: stop
    @.stopped:=true;
    if @.empty then
      -- wait for new part, then load
      waituntil station.occupied prio 1;
      station.cont.move(@);
    else
      -- wait until the station can receive the part
      -- then unload
      waituntil station.empty and
        station.operational prio 1;
      @.cont.move(station);
    end;
  end;
```

```

-- next step in the cycle
if sensorID = 1 then
  @.targetSensorID:=2;
else
  @.targetSensorID:=1;
end;
-- calculate direction: very simple
-- actual position is greater then
-- target position --> drive backwards
if self.~.sensorID(sensorID).position >
  self.~.sensorID(@.targetSensorID).position then
  @.backwards:=true;
else
  @.backwards:=false;
end;
--turn
@.stopped:=false;
end;
end;

```

Before starting the simulation, set the attribute targetSensorID of the robot in the class library to two. The robot should do its work without any further changes. If you need another robot in your simulation, use the track of the first robot (copy). The only things you need to change are the sensor positions and the objects in the sensorTable. Changes in programming are not necessary.

### 7.3.3 *Handling and Processing Times of the Robot*

Various approaches are exists to take into account the handling times of the robot.

- a) You can pause the robot for a while (e.g. after the part has been moved). The pause can be evaluated statistically (e.g. statTspPausingTime). If you take into account failures in the robot, then the statistics become slightly inaccurate since the pause overlaps the failures (you never reach your set availability). The robot can be paused with the statement <path>.startPause(number of seconds). Alternatively, you can pause the control of the robot with wait (<time>) for the duration of the handling time. In this case, however, you must record the necessary statistical data by yourself.
- b) You can calculate a speed of the robot that includes the handling time. When you have as default the motion and the handling time of the robot from one position to the next, you can use it to determine a speed between two sensors. A great advantage of this procedure is that the parts do not enter "too early" in the subsequent stations. In practice, the method is very accurate. If you need the handling time separately in the statistics, then you must record the handling time separately. At the end of an experiment, subtract the total handling time from the total driving time.

### Example: Robot Handling Time (Wait)

A robot unloads a non-accumulating conveyor line (e.g. transport chain). The subsequent process has an availability of 75 per cent and five minutes MTTR. In order to prevent interruptions of the continuous process (stop the chain), the robot stacks the parts into a buffer if the machine is unavailable and removes the parts on occasion again. Create a frame as in Fig. 7.29. Use, if possible, the objects from the previous examples.

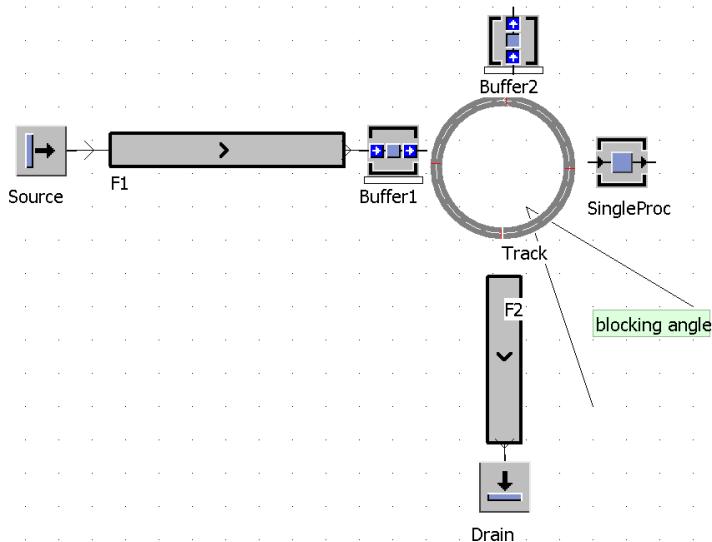


Fig. 7.29 Example Frame

Create the following settings:

Block	Attribute	Value
Source	Interval	70 sec.
	MU	Entity
F1	Length	7 meters
	Time	7 minutes
	Not accumulating	
Buffer1	Capacity	1
	Processing time	0
Buffer2	Capacity	100
	Processing time	0
SingleProc	Processing time	30 sec.
	Availability	75%
	MTTR	5 minutes
F2	Speed	1 meter/sec.
	Length	5 meters

The robot track needs a set of user-defined attributes like in Fig. 7.30.

Name	Value	Type
init		method
Robot	*.MUs.Robot	object
sensorControl		method
sensorlist		table
times		table

**Fig. 7.30** User-defined Attributes Track

The init method creates the robot on the track. The sensorlist stores the object references in relation to the SensorIDs. Place the sensors along the track as in Fig. 7.29 and fill the sensor list accordingly (Fig. 7.31). Assign the method sensorControl to each sensor.

	object 1
1	SingleProc
2	Buffer2
3	Buffer1
4	F2

**Fig. 7.31** Sensorlist

The table Times contains the necessary times of the robot from one position to another. The times are split into driving and handling times. Design the table like in Fig. 7.32 enter the values (in seconds).

	string 1	string 2	real 3	real 4
string	From	To	Movingtime	Handlingtime
1	Buffer1	SingleProc	3.00	8.00
2	SingleProc	Buffer2	1.50	8.00
3	Buffer2	SingleProc	1.50	8.00
4	SingleProc	F2	1.50	8.00
5	F2	Buffer2	3.00	0.00
6	F2	Buffer1	4.50	0.00
7	Buffer1	Buffer2	1.50	8.00
8	Buffer2	Buffer1	1.50	0.00
9	SingleProc	Buffer2	1.00	4.00

**Fig. 7.32** Table Times

The robot needs a user-defined attribute targetSensorID; the initial value is three (setting in the class library) and the initial direction of the robot is forward (backward = false). The sensorControl could (initially without consideration of the time) have the following content:

```
is
  station:object;
do
  -- get station
  station:=self.~.sensorList[1,sensorID];
  --stop at destination
  if sensorID = @.targetSensorID then
    @.stopped:=true;
    if @.empty then
      -- wait for part then load
      if sensorID=1 then
        waituntil station.resWorking=false prio 1;
      else
        waituntil station.occupied prio 1;
      end;
      station.cont.move(@);
    else
      -- wait until the station can get the part
      -- then unload
      if station.operational then
        @.cont.move(station);
        -- exception SingleProc
        --> store part if failure
        if sensorID=1 then
          waituntil station.resWorking=false or
            station.operational=false prio 1;
          if station.resWorking=false then
            station.cont.move(@);
          end;
        end;
      end;
    end;
  end;
  -- next step in cycle
  if sensorID = 1 then
    if @.empty then
      @.targetSensorID:=3;
    elseif station.operational then
      @.targetSensorID:=4;
    else
      @.targetSensorID:=2;
    end;
  elseif sensorID=2 then
    --buffer
    if @.occupied then
      --to SingleProc
      @.targetSensorID:=1;
    else
```

```

-- if the part is processed
--> empty to the singleProc
if singleProc.occupied and
    singleProc.process_finished then
    @.targetsensorID:=1;
else
    --get next part
    @.targetsensorID:=3;
end;
end;
elseif sensorID=3 then
    -- pick part from line
    if singleProc.operational and
        singleProc.empty then
        @.targetSensorID:=1;
    else
        @.targetSensorID:=2;
    end;
elseif sensorID=4 then
    -- empty
    if buffer2.occupied and buffer1.empty and
        singleProc.operational then
        @.targetsensorID:=2;
    else
        @.targetsensorID:=3;
    end;
end;
-- calculate direction
if self.~.sensorID(sensorID).position >
    self.~.sensorID(@.targetSensorID).position
then
    @.backwards:=true;
else
    @.backwards:=false;
end;
-- start driving
@.stopped:=false;
end;
end;

```

Considering Times Option 1: To map the handling times of the robot, pause the control of the robot for a while (command wait). The speed can be set directly in the transporter. In this example, three seconds is set for a rotation by 180°. For a track about 12 meters in length (radius of two meters), this means a speed of 2m/sec (six meters/three seconds). Search for the handling time at the station in the Times Table. As in the table above, a station may be assigned different handling times depending on the work to be executed and the following target.

Therefore, examine this in two steps. First, look in the first column for the current station. If the row does not contain the target station in the second column, then look further. Program an exit from the loop for the case that the requested combination is not present (otherwise the loop would run endlessly). If you have found the combination, read the handling time (and, in the next step, the travel time) from the table. The following part of the program should be located in the method `sensorControl` before `@.stopped: = false`. You need some additional variables:

```
i:integer;
actStation:string;
newStation:string;
rowNumber:integer;
found:boolean;
handlingTime:real;
movingTime:real;
```

Programming section for determining handling times from the table:

```
-- look for handling time
i:=1;
rowNumber:=0;
-- cursor to start
self.~.times.setCursor(1,i);
actStation:=self.~.sensorList[1,sensorID].name;
newStation:=
    self.~.sensorList[1,@.targetSensorID].name;
while i/=rowNumber loop
    -- look for the actStation in the first column
    found:=self.~.times.find({1,1}..{1,*},actStation);
    if found then
        --check whether the next station is located
        --in the second col --then set time
        rowNumber:=self.~.times.zeigerY;
        if self.~.times[2,rowNumber] = newStation then
            handlingTime:=
                self.~.times["Handlingtime",rowNumber];
            movingTime:=
                self.~.times["Movingtime",rowNumber];
        else
            --search next
            i:=rowNumber+1;
            if i > self.~.times.yDim then
                i:=rowNumber;
            else
                --set cursor for next search
                self.~.times.setCursor(1,i);
            end;
```

```

    end;
else
    i:=rowNumber;
end;
end;

--wait for handling time
wait(handlingTime);

```

The advantage of this approach is that it is relatively easy. But if you work with availability and failures, then this method can prove inaccurate. Wait does not take failures into account. This problem can be avoided by calculating the handling time in the movement-speed (time) of the robot (distance and time). If a failure occurs, the robot stops on the track; it starts again if the failure is over. Hence, the complete processing time is interrupted by failures. This approach has several advantages:

- You can easily represent different times for empty runs and drives under load.
- The part is transferred at the earliest after the driving and handling times.
- Failures and breaks interrupt the movement of the robot, so that an exact consideration of the temporal effects of failures and breaks is possible.

The disadvantages are more complex programming and a more difficult statistical analysis.

### Example: Robot Handling Time (Speed)

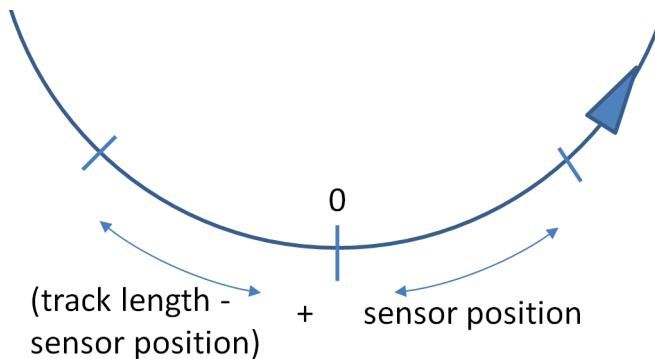
Copy the example “Robot handling time (wait),” e.g. from <http://www.bangsow.de>. Open the user-defined method attribute sensorControl of the robot track.

Preliminary considerations: The handling and driving time from one position to the next position in the cycle of the robot should be considered by the speed of the robot. Before driving the robot to the next position, the speed of the robot must be recalculated and set. The procedure at the beginning is similar to that in the previous example:

- The times must be found in the table; handling time and movement time are added together.
- The distance between the positions must be calculated (via the sensor position).
- From distance and time, the speed of the robot is calculated and the speed is assigned to the robot.
- The robot is started.
- For evaluation purposes, we will save the handling time in a user-defined variable.

When calculating the distance traveled by the robot, consider the following special case: The path is circular and it may be that the robot moves over the end of the track. The sensors are arranged linearly along the track. In such a case, you cannot determine the length of the track by subtracting the smaller from the bigger

position. As a condition, define the following: When the transporter travels forward and the position of the target sensor is smaller than the starting position or when the transporter is moving backward and the position of the target sensor is greater than the starting position, then the path length needs to be determined as follows: (track length - larger position) + smaller position (end + initial part, see Fig. 7.33).



**Fig. 7.33** Calculate Path Length

You need some additional variables in the sensorControl:

```
movingTime:real;
distance:real;
```

Insert a user-defined attribute into the robot in the class library: statHandlingTime (data type: time). In the loop in which the handling time is determined, add the following:

```
handlingTime:=self.~.times["Handlingtime",rowNumber];
movingTime:=self.~.times["Movingtime",rowNumber];
```

The determination of the distance between the sensors and the setting of the time might look like this (sensor control before @.stopped:=false):

```
-- take handling time into account
-- calculate the distance between two sensor
-- positions
if (@.backwards and (@.targetSensorID>sensorID)) then
    distance:=self.~.sensorID(sensorID).position +
        self.~.length -
        self.~.sensorID(@.targetSensorID).position;
elseif (@.backwards=false and (@.targetSensorID <
        sensorID)) then
    distance:=
        self.~.sensorID(@.targetSensorID).position +
```

```

        self.~.length -
        self.~.sensorID(sensorID).position;
else
    distance:=
        abs(self.~.sensorID(@.targetSensorID).position -
        self.~.sensorID(sensorID).position);
end;
-- recalculate the speed of the robot
@.speed:= distance/(handlingTime+movingTime);
-- record handling time
@.statHandlingTime:=@.statHandlingTime+handlingTime;
--start movement
@.stopped:=false;

```

You can now evaluate the proportions of the times of the robot at the total time:

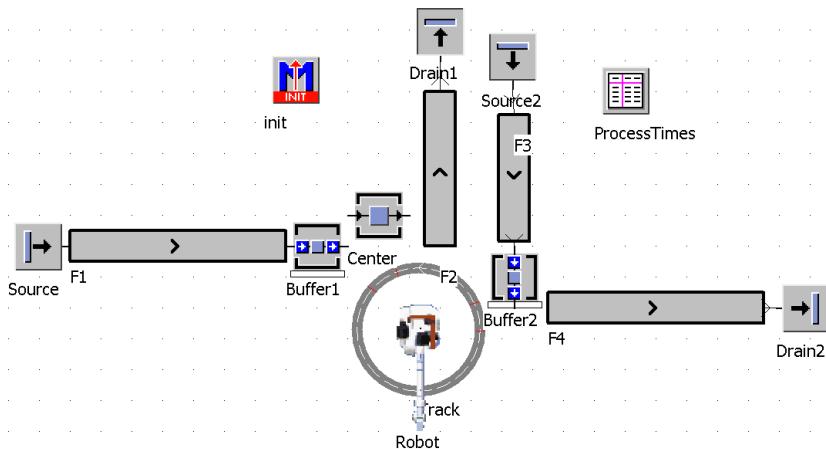
- Driving time (complete driving time – handling time)
- Handling time
- Waiting time

### 7.3.4 *Synchronous and Asynchronous Control of the Robot*

In principle, consider two approaches to programming robots in Plant Simulation. In the first variant, the robot has a waiting position. After each operation (the robot is empty), the robot moves to that position and waits for the next program call. In this way, the robot is able to process various tasks in any order (asynchronous). In the second variant, the robot is moved in a fixed cycle (synchronous). The robot processes one step after the other. At the end of the cycle, the robot is waiting in the first position at the start of the next cycle. The programming is much simpler here, since the cycle can be defined outside of the program. The differences in output can be serious in reality. Therefore, the control of the robot in the simulation has to be programmed in the same way as will happen in practice.

#### **Example: Robot—Synchronous and Asynchronous Control**

Your task is to simulate a handling robot. The robot loads and unloads a processing machine. The processing machine has varying processing times (approximately 50 seconds). Since the robot is less busy with the loading and unloading, it should also transfer and turn parts from another line. It is necessary to clarify to what cycle time of the second line this solution works. Create a frame like in Fig. 7.34.



**Fig. 7.34** Example Frame

Prepare the following settings:

Block	Attribute	Value
Source	Interval	80 seconds
	MU	Entity
Buffer1 / Buffer2	Capacity	1
	Processing time	0
Source2	Interval	60 seconds
Center	Processing time	Distribution dEmp TableFile ProcessTimes

To model the varying processing time, we use a frequency table. This type of data analysis can be carried out in many cases using existing data. Fill the table processing times with frequencies as shown in Fig. 7.35.

	time 1	real 2
string	Time	Frequency
1	40.0000	10.00
2	50.0000	70.00
3	1:00.0000	10.00
4	1:10.0000	5.00
5	1:20.0000	5.00

**Fig. 7.35** dEmp Frequency Table

The center and the buffer have to signal completion or delivery via a user-defined attribute (hereinafter process\_finished). This attribute should be set to true

in an exit-control (front). After unloading the stations, this attribute is set to false. You can also take this route if you want to use the conveyor lines without buffers. In order not to run the risk to transfer the part before the end of the conveyor line (to "beam"), wait until the part has reached the end of the conveyor line. Insert in the track of the robot a user-defined method attribute (sensorControl). Create sensors on the track like in Fig. 7.36.

ID	Position	Front	Rear	L.	Path
1	3.1m	x			self.sensorControl
2	4m	x			self.sensorControl
3	5.5m	x			self.sensorControl
4	7m	x			self.sensorControl
5	8m	x			self.sensorControl

**Fig. 7.36** Robot Track Sensors

To easily translate the sensor position in an object reference, we use a sensor table (user-defined attribute of the track). Format the table as shown in Fig. 7.37 and enter the appropriate values.

	object 1
1	root.F4
2	root.Buffer2
3	root.F2
4	root.Center
5	root.Buffer1

**Fig. 7.37** Sensor List

Additionally, we need to program into the cycle an attribute cycle\_position (integer) and a table cycle\_table (user-defined attributes track). In the beginning, the cycle of the robot should appear as follows: The robot waits at Buffer1 for a part and moves with the part to the machining center. There, it unloads the part and moves empty to Buffer2. It waits for the part, then collects and loads the part, and turns to the line F4. There, it unloads the part and turns to the machining center. The robot waits until the processing of the part is complete. Then it loads the part and turns to the conveyor line F2. After unloading the part, the cycle starts again at Buffer1. The information that you need for programming can be stored in a tableFile as shown in Fig. 7.38 (table cycle\_table as a user-defined attribute in the track of the robot).

	integer 1	integer 2	real 3	boolean 4
string	ID	DestinationID	Time	backwards
1	5	4	10.00	true
2	4	2	7.00	true
3	2	1	6.00	true
4	1	4	6.00	false
5	4	3	10.00	true
6	3	5	6.00	false

**Fig. 7.38** Cycle Table

The line number is the index of the cycle step. The table can be read as follows: Step 1 starts on Sensor 5 (see sensor table). The next destination is Sensor 4, where handling and driving last a total of 10 seconds and the robot has to move backwards to the target. The user-defined attribute `cycle_position` stores between the different calls of the method, the current position in the `cycle_table`. Insert in the robot a user-defined attribute `TargetSensorID` (integer).

Method `init` (user-defined attribute `track`): The `init` method has to set the `cycle_position` to 1, to create the robot and set the `targetSensorID` to the first position in the `cycle` (`cycle_table[1,1]`). Therefore, the method should have the following content:

```
is
  robot:object;
do
  --create robot
  robot:=.BES.Robot.create(self.~);
  --start new cycle
  self.~.cycle_position:=1;
  -- first destination
  robot.targetSensorID:=self.~.cycle_table[1,1];
end;
```

Preliminary considerations for sensor control: The sensorControl system is equal at each position. If the robot is empty, it waits until a part is available (and the station has completed the processing). It then loads the part. If the robot arrives occupied at the sensor, it must wait until the station is operational and empty, before loading the part on the station. Thereafter, a new destination is set for the robot (`targetSensorID`). The direction and speed of the robot is reset (to take account of the times), after which the robot is set in motion again. If there are no exemptions from the cycle, a sensor control might look like this, no matter how complex the cycle is:

```
(SensorID : integer)
is
  station:object;
```

```
_time:real;
distance:real;
do
  if sensorID= @.targetSensorID then
    -- stop at the destination
    @.stopped:=true;
    station:=self.~.sensorList[1,sensorID];
    --if empty wait for the first part
    if @.empty then
      waituntil station.occupied and
        station.process_finished prio 1;
      station.cont.move(@);
      station.process_finished:=false;
    else
      --occupied
      waituntil station.operational prio 1;
      @.cont.move(station);
    end;
    -- set new target
    @.targetSensorID:=
      self.~.cycle_table["DestinationID",
      self.~.cycle_position];
    --set direction
    @.backwards:=self.~.cycle_table["backwards",
      self.~.cycle_position];
    -- calculate speed
    _time:=self.~.cycle_table["Time",
      self.~.cycle_position];
    distance:=abs(self.~.sensorID(sensorID).position-
      self.~.sensorID(@.targetSensorID).position);
    @.speed:=distance/_time;
    --drive
    @.stopped:=false;
    --next cycle
    self.~.cycle_position:=self.~.cycle_position+1;
    if self.~.cycle_position >
      self.~.cycle_table.yDim then
      self.~.cycle_position:=1;
    end;
  end;
end;
```

You can now use the robot for different applications (you only need to define the sensors corresponding to the sensor list and adjust the cycle\_table). Programming changes are not necessary.

Note: If you need to load with the robot onto a pallet or a vehicle, extend the cycle\_table by one column (e.g. name: content; data type: Boolean). Set the value in this column to true in case you need to load the part onto the content of the station. This can easily be added to the sensor control.

The robot in the above example fails in turning the required number of parts from the conveyor line F3. The robot would do this if it had the opportunity to meet a decision after unloading the part to F4, transferring an additional part (if one exists) or turning to the machining center. For this, the robot requires another hardware configuration that allows it to receive information and evaluate this information. The programming of the robot is also more extensive. Duplicate the frame in the class library. With the duplicate, you can program a variant. The decision of the robot might look like this (sensor control):

```
(SensorID : integer)
is
  station:object;
  _time:real;
  distance:real;
do
  if sensorID= @.targetSensorID then
    -- stop at the destination
    @.stopped:=true;
    station:=self.~.sensorList[1,sensorID];
    --if empty wait for the first part
    if @.empty then
      waituntil station.occupied and
        station.process_finished prio 1;
      station.cont.move(@);
      station.process_finished:=false;
    else
      --occupied
      waituntil station.operational prio 1;
      @.cont.move(station);
    end;
    -- set new target make decision
    if sensorID=1 and center.process_finished=false
      and buffer2.process_finished then
      --another part from buffer2
      @.targetSensorID:=2;
      @.backwards:=false;
      _time:=self.~.cycle_table["Time",
        self.~.cycle_position];
      --jump back in the cycle one position
      self.~.cycle_position:=self.~.cycle_position-1;
    else --standard process
      @.targetSensorID:=
```

```
        self.~.cycle_table["DestinationID",
        self.~.cycle_position];
--set direction
@.backwards:=self.~.cycle_table["backwards",
    self.~.cycle_position];
_time:=self.~.cycle_table["Time",
    self.~.cycle_position];
self.~.cycle_position:=
    self.~.cycle_position+1;
if self.~.cycle_position >
    self.~.cycle_table.yDim then
    self.~.cycle_position:=1;
end;
end;
-- calculate speed
distance:=abs(self.~.sensorID(sensorID).position-
    self.~.sensorID(@.targetSensorID).position);
@.speed:=distance/_time;
--drive
@.stopped:=false;
--next cycle
end;
end;
```

With this adjustment, the object is attained by the robot.

## 7.4 The LockoutZone

Starting with Version 10, Siemens has inserted the LockoutZone in the resource objects. You can use this to represent simple protection circuits. As a protection circuit, you can define a set of machines that is unable to work, when one of them has malfunctions. This is necessary, e.g. when carrying out repairs in interlinked equipment and is achieved by the possibility that all material flow blocks can be stopped (as there is a statistical value). The state stopped works similarly to failures, as all machining operations will be interrupted. Only when all stations in the protection circuit are in an undisturbed state, are all stations released. In an already begun processing, only the remaining time is used as the processing time.

### Example: LockoutZone

You are intended to simulate a typical case for a protection circuit. Two machines are loaded and unloaded with the help of PickAndPlace robots. The entire cell is surrounded by a fence. In order to perform repairs, all PickAndPlace robots and all machines must be shut down. Create a frame according to Fig. 7.39.

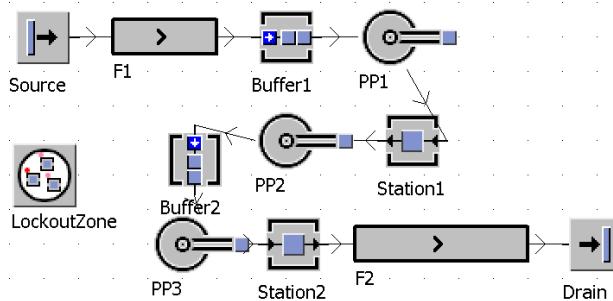


Fig. 7.39 Example Frame

The source produces one part per minute; Station 1 and Station 2 each have a processing time of 45 seconds. The PickAndPlace robots have an availability of 95 per cent and the machines of 85 per cent with MTTR of 20 minutes. The buffers each have a capacity of 10 parts at a processing time of 10 seconds. To activate the LockoutZone, insert all stations (tab Objects) that are interacting in case of a failure. Turn off the inheritance (click on the green square) and then add to the list by dragging and dropping items (Fig. 7.40).

Objects	
1	.Modelle.frame.PP1
2	.Modelle.frame.Station1
3	.Modelle.frame.PP2
4	.Modelle.frame.PP3
5	.Modelle.frame.Station2

Fig. 7.40 Objects LockoutZone

In the Controls tab, determine when the stations are to be stopped (Fig. 7.41).

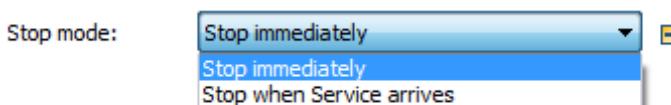


Fig. 7.41 LockoutZone Stop Mode

- Stop when service arrives (if you request repair resources only when the worker arrives at the machine)
- Stop immediately (all remaining machines will be stopped at once)

Run the simulation for a while. Then, open the Statistics tab for Station2. You can see the stopped portion in the machine statistics (Fig. 7.42).

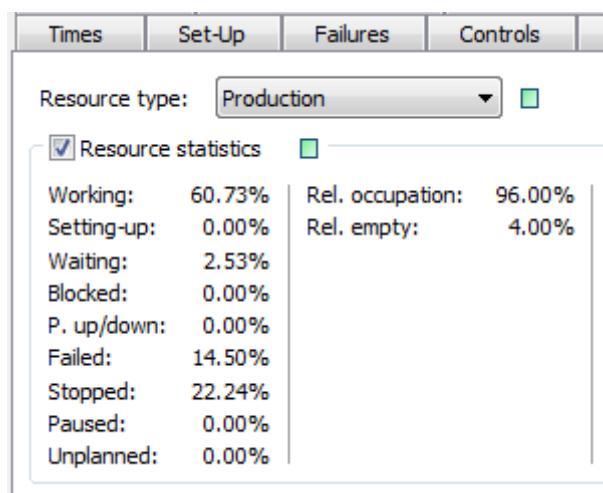


Fig. 7.42 Statistics

Note: The LockoutZone has a strong influence on the output. Especially when you insert a large number of machines within the LockoutZone, a relatively strong reduction of the system output results (depending on the availability of the individual stations). Inform yourself about the real used control strategies and repair scenarios; otherwise, the simulation will give inaccurate results.

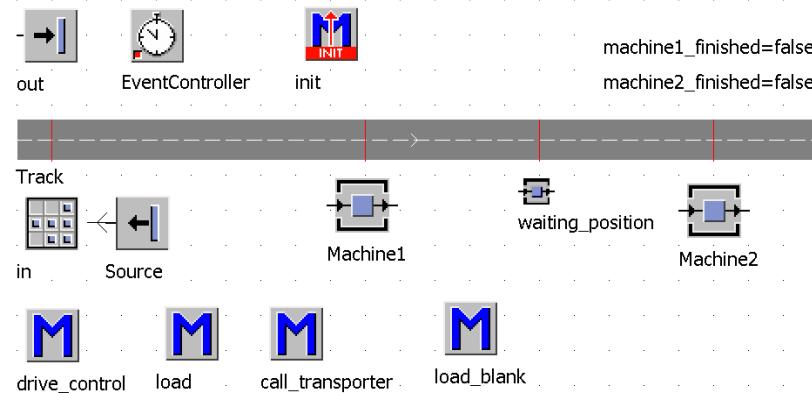
## 7.5 Gantry Robots

Gantry or linear robots are often used for loading and unloading machines. Here, the robots move along a straight portal. The travel path is mostly limited by a cable package. In principle, one must distinguish the gantry loader on the number of grippers. Generally, we can distinguish in I-loader (only one gripper) and H-loaders (two grippers). The number of the grippers is essential for designing the control of the gantry robot. Gantry robots can be simulated very simply by the use of the transporter and track classes. The simplest variant is the simulation of the gantry robot without modeling the grippers. The time it takes the gripper for loading and unloading can be considered as handling time analogous to the handling robots. In such cases, the robot or loader can be parameterized very well to model standard situations.

### Example: Portal Loader—Parallel Processing

You are to simulate a portal loader that loads two machines. The machines simultaneously process the same kind of part. The loader picks up parts at a place

at the beginning of the track parts (capacity one part) and distributes them to the machines. If both machines are loaded and working, the loader waits empty between the two machines. When the first machine has completed processing, the loader drives to the machine and unloads it. A time of five seconds for the handling by the loader is considered (the movement in the z-axis is not simulated). Create a frame like in Fig. 7.43.



**Fig. 7.43** Example Frame

Settings: The source produces parts at intervals of 3:30 minutes. The processing time of Machine1 is 7:50 minutes, of Machine2 7:40 minutes. Both machines have an availability of 90 per cent and an MTTR of 45 minutes. The track has a length of 25 meters while the transporter has a length of 1.5 meters, a speed of 1 m/s and a capacity of one part. Insert the sensors into the track (as the `drive_control`) according to Fig. 7.44.

ID	Position	Front	Rear	L.	Path
1	1m		x		<code>drive_control</code>
2	10m	x			<code>drive_control</code>
3	15m	x			<code>drive_control</code>
4	20m	x			<code>drive_control</code>

**Fig. 7.44** Sensors Portal Loader

The global variables `machine1_finished` and `machine2_finished` have the data type Boolean and the initial value false.

1. Insert the transporter: The init method creates the transporter. Prior to that, all MUs will be destroyed.

Method init:

```
is
do
  deleteMovable;
  .MUS.Transporter.create(track,12);
  track.cont.destination:=in;
  track.cont.backwards:=true;
end;
```

2. Program the driving control: The transporter will be addressed for each trip. For this, the attribute destination is used. Destinations are assigned to certain sensor IDs. Once the transporter arrives at the destination (target sensor ID), the transporter stops. The method drive\_control needs a parameter for the sensor\_ID. First, the transporter is to stop at the sensor\_ID 1.

Method drive\_control:

```
(sensorID : integer)
is
do
  if sensorID=1 and (@.destination=in
  or @.destination=out) then
    @.stopped:=true;
  end;
end;
```

3. Program the method for loading the unfinished part: The method load\_blank uses the transporter as the parameter and should have the following functionality: The method waits until the place "in" is occupied. If Machine1 or Machine2 is empty, the transporter loads the part and drives to the empty machine. If both machines are occupied, the transporter drives to the waiting position. The handling time is taken into account by pausing the transporter.

Method load\_blank:

```
(transporter:object)
is
do
  --search an empty and operational machine
  --wait for an unfinished part
  waituntil in.occupied prio 1;
  if machine1.empty and machine1.operational then
    in.cont.move(transporter);
    transporter.destination:=machine1;
  elseif machine2.empty and machine2.operational
    then
    in.cont.move(transporter);
    transporter.destination:=machine2;
```

```

else
  transporter.destination:=waiting_position;
end;
--drive forward
transporter.backwards:=false;
-- 5 seconds handling time
transporter.startPause(5);
transporter.stopped:=false;
end;

```

Call within the method drive\_control: If a loaded transporter arrives at Sensor 1, the part is transferred to the drain "out". Then the method load\_blank is called.

Method drive\_control:

```

(sensorID : integer)
is
do
  if sensorID=1 and (@.destination=in or
                     @.destination=out) then
    @.stopped:=true;
    if @.empty then
      load_blank(@);
    else
      -- move parts to the drain
      @.cont.move(out);
      --load new parts
      load_blank(@);
    end;
  end;
end;

```

4. Program the method load: The loaded transporter is to stop at the machine and load the part onto the machine (if the machine is operational). Thereafter, the transporter moves to the waiting position. If the transporter is empty, the transporter unloads the machine and drives to the "out" drain. The method load requires three parameters: machine, transporter and the direction of the waiting position (backwards true/false). The method load could look like this:

```

(transporter:object;machine:object;
directionWaitingPosition:boolean)
is
do
  transporter.stopped:=true;
  if transporter.occupied then
    -- transporter is loaded
    waituntil machine.operational prio 1;
    transporter.cont.move(machine);
    transporter.destination:=waiting_position;
    transporter.backwards:=

```

```

        directionWaitingPosition;
else
    --transporter is empty
    machine.cont.move(transporter);
    transporter.destination:=out;
    transporter.backwards:=true;
end;
-- start after 5 seconds
transporter.startPause(5);
transporter.stopped:=false;
end;

```

5. Program the method call\_transporter: If the machines are ready, they must send a signal. The class SingleProc provides the method Ready that returns true if the station has finished processing parts. This value, however, is not observable. One solution would be the following: If the machine is ready, the processed part triggers a control that sets a global variable to true (e.g. machine1\_finished). Global variables are observable and an appropriate action can be triggered. Within the frame, the ready variables consist of the machine name and “\_finished”. A universal method for registering the finished machines could look like this:

Method call\_transporter:

```

is
do
    -- ? object, that calls
    str_to_obj(? .name+_finished) :=true;
end;

```

str\_to\_object converts a string (object name) to an object reference. Assign the method call\_transporter to the exit control (front) of Machine1 and Machine2.

6. Complete the method drive\_control: Within the drive\_control, the method load must be called at the positions of Machine1 and Machine2. A control for the waiting position is established at the position of the waiting position: If Machine1 or Machine2 is operational and empty, the transporter drives to the station and delivers a new part; otherwise, the transporter is waiting until Machine1 or Machine2 is ready. The transporter might change its direction, set a new destination and set the finished variable of the machine to false. Finally, the transporter drives to the machine. The completed method drive\_control should look like this:

```

(sensorID : integer)
is
do
    if sensorID=1 and (@.destination=in or
        @.destination=out)
    then
        @.stopped:=true;
        if @.empty then

```

```
    load_blank(@);
else
    -- move parts to the drain
    @.cont.move(out);
    -- load new parts
    load_blank(@);
end;
elseif sensorID=2 and @.destination = machine1
then
    load(@,machine1,false);
elseif sensorID=4 and @.destination = machine2
then
    load(@,machine2,true);
elseif sensorID=3 and @.destination =
    waiting_position then
    stopped:=true;
    --new part if one machine is empty and
    --operational
    if (machine1.empty and machine1.operational)
        or
        (machine2.empty and machine2.operational)
    then
        @.destination:=in;
        @.backwards:=true;
    else
        -- wait until one machine has finished
        waituntil machine1_finished or
        machine2_finished
        prio 1;
        if machine1_finished then
            @.destination:=machine1;
            @.backwards:=true;
            machine1_finished:=false;
        elseif machine2_finished then
            @.destination:=machine2;
            @.backwards:=false;
            machine2_finished:=false;
        end;
    end;
    @.stopped:=false;
end
end;
```

# Chapter 8

## Warehousing and Procurement

You can use several objects for modeling storage and warehouse processes. The main building blocks are buffer and store. A number of tasks of material flow simulations deal with the simulation of buffer, warehousing and transport operations. Questions to be addressed, for example, include the dimensions of buffers or warehouses and the review of the planned transport capacity. The modeling of warehouse controls, combined with production control systems, is a major challenge for the simulation.

### 8.1 Buffer

Plant Simulation distinguishes between two types of buffers: PlaceBuffer and Buffer.

#### a) PlaceBuffer

The MUs pass the PlaceBuffer one after another in the "processing time." MUs cannot pass each other within the buffer. Only when the MU has reached the place with the highest number can it be passed on. When the last MU has been passed on, all other MUs can move forward one place. The processing time can be specified only in relation to the entire buffer (e.g. dwell time in the buffer is 20 minutes), not in relation to a single place (10 places per two minutes). The attribute Accumulating determines whether the exit of the buffer is blocked (e.g. the successor is occupied) and any following MUs move up (Accumulating = TRUE) or have to wait.

#### b) Buffer:

The buffer does not have a place-oriented structure. After the processing time is over, you can remove the MU again. You can determine a mode for unloading:

- Buffer type Queue: First in, first out
- Buffer type Stack: Last in, first out

From Version 10, you can represent the filling level of the buffer graphically below the block icon (tab Attributes). As an MU exits the buffer, the following blocks are served in turn (or special exit behavior). The most important settings are capacity (number of places in a buffer), processing time (total time, the dwell time of a piece in the buffer), recovery time and cycle time.

## 8.2 The Sorter

If you have to map queues controls, then the sorter can be helpful.

### 8.2.1 Basic Behavior

The sorter can receive a certain number of MUs and pass them on in a different order. The removal order of MUs, which the sorter contains, depends on definable priorities. The MU with the highest priority will be transferred first, regardless of when it entered.

E.g. the following selection criteria are offered:

- Duration of stay
- MU attribute
- Control

The content of the sorter is sorted, if either

- an MU enters the sorter or
- the content of the sorter is accessed.

When several MUs have the same value within a sort criterion, then the order of these MUs remains undefined. You can use the sorter for simulating queue logics.

### 8.2.2 Attributes of the Sorter

A number of rules (e.g. shop floor management) exist for controlling queues. An important criterion is, for example, the throughput time of an order (from entry to the production and, finally, to delivery to the customer). Special orders from major customers are often preferred in order to deliver quicker. The throughput time of the remaining orders thus increases. Another rule is that the order with the least setup cost will be fulfilled first. The simplest queue management follows the first come, first served principle (or first in, first out).

Fig. 8.1 shows the main attributes of the sorter.

Attributes	Times	Failures
Capacity:	4	<input type="button" value=""/>
Order:	Ascending	<input type="button" value=""/>
Time of Sort:	On Entry	<input type="button" value=""/>
Sort criterion:	Occupation Time	<input type="button" value=""/>

**Fig. 8.1** Sorter Attributes

**Capacity:** Enter the number of stations in your sorter. "-1" stands for unlimited capacity. You can access individual stations by their index ([...]).

**Order:** The sort order determines whether the MUs are sorted in ascending or in descending order. (Priority 1 very high).

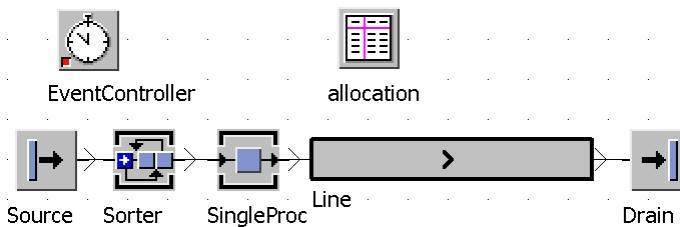
**Time of Sort:** When should sorting occur? If On Entry is selected, newly entering MUs will be sorted into the existing order of the other MUs. The sequence is not updated, even if the values of the sort criteria change. When you select the option On Access, the MUs will be sorted dynamically. At each entrance of a new MU or before moving it, the MUs will be reordered (taking into account the current values of the sort criterion).

**Sort Criterion:** Sort criteria can be:

- Occupation Time: The MUs will be sorted according to their occupation time in the sorter (descending: first in, first out; ascending: first in, last out)
- MU Property: You can enter order attributes and statistical values (statistical values only if statistics for the MUs are active).

### Example: Sorter

A process with an availability of 50 per cent will be simulated. A sufficiently large buffer is located in front of the process. The parts will be processed after blockages with different priorities. We want to achieve that parts with higher priorities have a much lower throughput time than the parts with lower priority. Create a folder color\_sorting below models. Create a frame within the folder color\_sorting (right-click the folder icon—New—Frame). Duplicate all the required objects in this folder. Create a frame like in Fig. 8.2.



**Fig. 8.2** Example Frame

Insert three entities, and name them red, green and blue. Assign them different colors (recommended: 5x5 pixels, colors according to the names) to distinguish among them. Open the source and set the following values: interval: constant two seconds; MU selection: random; table: allocation. Enter the values from Fig. 8.3 into the table allocation (Note: You can insert the addresses using drag and drop. Drag the relevant parts from the class library to the table, and drop them there):

	object 1	real 2	stri 3
string	MU	Frequencies	Name
1	.MUs.red	10.00	
2	.MUs.green	60.00	
3	.MUs.blue	30.00	

**Fig. 8.3** Table Allocation

SingleProc: Processing time: one second; availability: 50 per cent; 30 minutes MTTR (based on the simulation time); line: length of eight meters; 1 m/s speed, accumulating; drain: zero seconds processing time. Create a user-defined attribute for each part (Double-click the part in the class library and then select the tab User defined attributes, Fig. 8.4):

Name	Value	Type	C.
importance	1	integer	*

**Fig. 8.4** User-defined Attribute

Set the following values for the attribute "importance": red: 1, blue: 2, green: 3. The parts in the sorter should be sorted according to the attribute "importance." In the sorter, select the criteria by which it sorts (default on entrance of a new part into the sorter, Fig. 8.1 Fig. 8.5).

Attributes	Times	Failures	Controls	Exit Strategy
Capacity:	10000			
Order:	Ascending			
Time of Sort:	On Entry			
Sort criterion:	MU-Property		User-defined Attribute	
			importance	
<input type="button" value="Start Sorting"/>				

**Fig. 8.5** Sorter Settings

Sort Order: ascending—the minimum value of the importance jumps forward in the queue and is sorted in ascending order; sort criterion: MU-Property—User-defined Attribute—"importance." By clicking on Start Sorting you can start the sorting process. Run the simulation for a while. If a failure occurs, you will see a

series of red parts exiting the SingleProc (the parts enter the sorter in mixed order). Look at the type-specific statistics of the drain. Click the button Detailed Statistics Table on the tab Type Specific Statistics (Fig. 8.6). The value LT\_Mean shows the average throughput time of the parts. The part red has a much lower throughput time than the part green.

Type Statistics					
	Type	Time	Total through	%Parts	LT_Mean
1	blue	39:22:18:35.	514899	30.15	12:12.5835
2	green	39:22:18:36.	1021826	59.83	4:57:34.4536
3	red	39:22:18:31.	171221	10.02	7:42.1554

Fig. 8.6 Statistics of the Drain

### 8.2.3 Sort by Method

If the sorting criterion is too complex, you can use a method to calculate a value for the sorting criterion of the MUs in the sorter. Plant Simulation calls this method at the time of sorting for each part and then sorts the MUs after the returned value.

#### Example: Sort by Method

To use the sorter to create a particular sequence of different parts, create a frame like in Fig. 8.7.

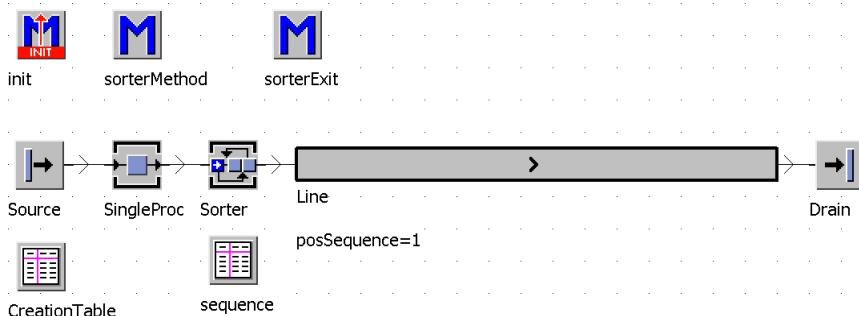


Fig. 8.7 Example Frame

Create three parts (part1, part2, part3) in the class library and color the icons differently. The source produces these parts consecutively in batches of 100 pieces (MU-selection: sequence cyclically; table: CreationTable; interval: three seconds). Set the procTime of the SingleProc to three seconds. The method sorterExit is the exit control (rear) of the sorter and increases the variable posSequence or sets it to one for the beginning of a new sequence. The method sorterExit has the following content:

```

is
do
  posSequence:=posSequence+1;
  if posSequence > sequence.yDim then
    posSequence:=1;
  end;
end;

```

The init method blocks the exit of the sorter object for one hour to create a stock for sequencing:

```

is
do
  sorter.exitLocked:=true;
  wait(3600);
  sorter.exitLocked:=false;
end;

```

Prepare the following settings in the sorter:

- Capacity: 1,000
- Order: Descending
- Time of Sort: On access
- Sort criterion: Method
- Method: sorterMethod

The sorter should transfer parts according to a sequence defined in the table Sequence. Create in this table a sequence consisting of the names of the MUs (row by row in the first column—for example: Part1, Part2, Part3). The method now checks for each part if the name of the part corresponds to the current sequence position. If yes, then 100 is returned; if not, zero is returned. The sorterMethod could look like this:

```

: real
is
do
  --return 100 if this is the right part for
  --the sequence, otherwise 0
  if @.name=sequence[1,posSequence] then
    return 100;
  else
    return 0;
  end;
end;

```

### 8.3 The Store, Warehousing

There are many reasons for warehousing in practice. These include:

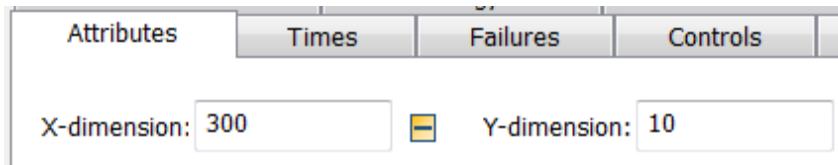
- Mismatch in the timing of production and consumption
- Uncertainty relating to the exact required quantity

- Speculation (price reduction or price increase)
- Function within the production process (e.g. drying, aging, etc.)

According to the various functions of the warehouse, there are numerous practical design options. With the store, Plant Simulation provides a matrix-oriented block for modeling warehouse processes.

### 8.3.1 *The Store*

The store has an unlimited number of storage places that are organized in a matrix (Fig. 8.8). As long as one place is free, the store can receive MUs. Without a control method, the store places the MU on any free place in the matrix.



**Fig. 8.8** Store Attributes

As opposed to the active material flow objects, the store has no setup time or processing time and no exit controls. The MUs remain in the store until their removal by using a control. When you want to reduce the dimension of a store, you can only eliminate empty places. If MUs are still located on the places you want to delete, you will first have to delete the MUs or transfer them to other places within the smaller dimension. If the store has failed, it cannot receive MUs, but can move MUs out of it.

### 8.3.2 *Chaotic Warehousing*

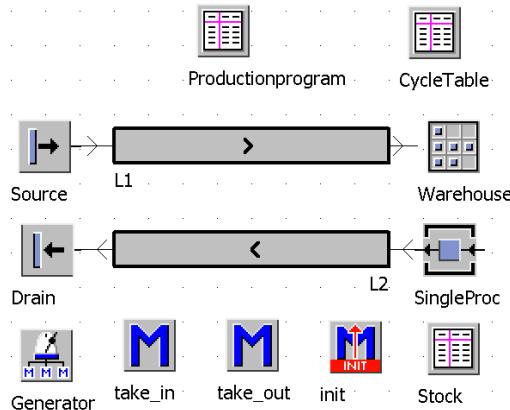
The store block requires some adjustments to enable it to simulate the work of a warehouse in a realistic way. In principle, a distinction is made in systematic and chaotic warehousing. In systematic warehousing, each part is assigned to a fixed place. But in chaotic warehousing, the part is stored at the next available warehouse place. In this case, the warehouse must maintain a list in which the storage is indicated (with part and warehouse place). The following tasks of a chaotic warehousing should be handled in the following:

- Pre-assignment of the warehouse (initial stock)
- Find the next free warehouse place
- Storage of parts, creating an inventory list
- Finding parts in stock (via the inventory list)
- Retrieval of parts (e.g. FiFo)

For an effective modeling of similar facts, the store block of the Plant Simulation class library should be equipped with appropriate methods.

### Example: Warehouse Simulation (Version TableFile)

A company produces parts in lots of 20 pieces. The parts are individually stored at the next available warehouse place. Three different parts are produced one after another. The parts will be shipped from the warehouse individually in a sequence (1, 1, 1). The warehouse has a capacity of 100 pieces (10x10). The average cycle time is one minute. For demonstration of the facts, a simple frame as shown in Fig. 8.9 is sufficient.



**Fig. 8.9** Example Frame

Create the following settings:

Block	Attribute	Value
L1 / L2	Length	8 meters
	Speed	0.008 meters/sec.
	Accumulating	true
SingleProc	ProcTime	45 seconds
Part1, Part2, Part3	Length, width	0.4 meters
Stock (Store)	x-dim, y-dim	10
Source	Interval	1 minute
	MU-Selection Table	Production program

Insert three parts in the class library and color them differently. Set the icon-size to 11x11 pixels. The table production program requires the entries in Fig. 8.10.

	object 1	integer 2
string	MU	Quantity
1	.MUs.Part1	20
2	.MUs.Part2	20
3	.MUs.Part3	20

**Fig. 8.10** Production Program

### 8.3.2.1 Inventory, Process of Storage

A big problem of chaotic warehousing is that a fixed storage location is not assigned to the individual parts. After storing a part, there are two ways to locate the part again:

- The entire warehouse is systematically searched. If the part is found, it is removed from the warehouse. This variant is rarely found in practice.
- When the part is stored, a list is written. In this list, the part (name) and the storage location are recorded (with storage aisle and place coordinate x, y). When the part is requested, the list is searched for the part and the part is removed from the warehouse.

### 8.3.2.2 Structure of the Inventory (Table Stock)

The inventory in the example above must have at least three columns: part name, x-position, y-position. The corresponding data types are: string, integer and integer. Format the table stock like in Fig. 8.11.

	string 1	integer 2	integer 3
string	Name	X	Y
1			

**Fig. 8.11** Table Stock

The storage process must consist of three steps:

1. Looking for a free place
2. Move the part to this place
3. Record the part in the table stock

### 8.3.2.3 Looking for a Free Place

The search for a free place must be created as a function of the warehouse. The function does not require a transfer parameter, but should return two results (one x

and one y coordinate). A function can return only one value as a result. To solve this problem, there are at least three approaches:

- Return of the x, y coordinate as a reference (ByRef)
- Return of the coordinate as a list
- Storing the coordinate in user-defined attributes of the store block

#### *Passing of Parameters as Reference*

Normally, Plant Simulation makes copies of the transfer parameters; within a function, you will work with the copies. The values of the variables in the calling method remain unchanged by the function call. But you can also pass references to variables. In this way, you can change the value of the passed variable within the function. If you pass several variables by reference, then several values can be changed simultaneously by a function.

#### **Continuation of the Example**

Insert in the warehouse a user-defined attribute of type method. Name it “getFreePlace.” The function must search through all the places of the warehouse. At the first free place, the method stops and the coordinates of the location are stored in the transfer parameters. If no free place exists, false is returned and the variables remain unchanged. The attributes and methods from Table 8.1 are needed.

**Table 8.1** Attribute and Methods of the Store

Method/Attribute	Description
<path>.xDim	Maximum x-dimension of the warehouse (columns)
<path>.yDim	Maximum y-dimension of the warehouse (rows)
<path>.pe(x,y)	Access to the place with the coordinate x,y
<path>.full	Returns true, if the warehouse is full

**Note:** If you use a method attribute, then for accessing the methods and attributes of the corresponding object, you need to refer to the object with self.~.

If you want to check whether the storage is full, then the command in a method attribute is: self.~.full. The method for determining a free place might look like this:

```
(byRef x,y:integer) :Boolean
is
  i:integer;
  k:integer;
do
  if self.~.full then
    return false;
```

```

else
  for i := 1 to self.~.xDim loop
    for k := 1 to self.~.yDim loop
      if self.~.pe(i,k).empty then
        --save coords into parameters
        x:=i;
        y:=k;
        return true;
      end;
    next;
  next;
end;
end;

```

### 8.3.2.4 Store and Register Parts

The call is made at the end of the conveyor line (exit control). The storage process must be recorded in a table (stock). You need the methods of the table from Table 8.2 to write the table stock.

**Table 8.2** Methods of the TableFile

Method	Description
<path>.yDim	Returns the number of entries in the table
<path>.writeRow(x,y,values)	Writes a row in the table; the next free row can be found at the y-position yDim+1

In the method take\_in, first an empty place is searched. Then, the arrived part is moved to the empty place. At the end, the name of the part and the x- and y-coordinates are written into the table. To ensure the success of the storage, the method waits at the beginning until the warehouse has at least one empty place (full = false). The method take\_in needs the following programming:

```

is
  x:integer;
  y:integer;
  partName:string;
do
  waituntil warehouse.full=false prio 1;
  --look for free place
  warehouse.getFreePlace(x,y);
  partName:=@.name;
  @.move(warehouse.pe(x,y));
  --register
  stock.writeRow(1,stock.yDim+1,partName,x,y);
end;

```

### 8.3.2.5 Find a Part and Remove It from the Warehouse

Every minute, one part should be removed from the warehouse. The basis for the retrieval is a cycle table. After this cycle table, the parts are searched for in the stock and moved out of the warehouse. A simple solution of this task would be the use of a generator and a method. The generator calls the method at regular intervals. In the method, the delivery is realized in the correct order. The methods and attributes of Table 8.3 are necessary:

**Table 8.3** Attributes and Methods for Searching in TableFiles

Attribute/Method	Description
<path>.find([range], value)	Finds a value within a range. The method returns true, if the process was successful
<path>.setCursor(x,y)	Sets the cell cursor in the table to the passed coordinate
<path>.cursorY	Reads and sets the row position of the cursor in the table
<path>[x,y]	Reads the value from the cell position x, y
<path>.cutRow(y)	Removes the row y from the table and all other entries move up
<path>.yDim	Returns the row number of the last row with content

#### *Searching in TableFiles*

Plant Simulation uses a cursor concept for positioning within a table. Before starting a search process, move the pointer with the method `setCursor` to an initial position (for example, in the first cell). If the search was successful, Plant Simulation sets the pointer to the result position. Then, with the help of the cursor (`cursorY`), you can read the position of the cell of the hit. If the search is unsuccessful, the method "find" returns false.

#### **Continuation of the Example**

Insert the data from Fig. 8.12 into the table `CycleTable`.

	string
1	
string	MU_Name
1	Part1
2	Part2
3	Part3

**Fig. 8.12** CycleTable

The position within this cycle must be saved between the calls of the method. This can be realized as an attribute within the `CycleTable`. Insert a user-defined attribute in the `CycleTable` (Tools—User-defined Attribute—name: `position`; data type: integer). The method `take_out` must first determine the part according to the

position in the cycle. Then, the part must be found in the stock table and transferred from the warehouse to the SingleProc. The row in the stock table must be deleted after removing the part from the warehouse. Method take\_out:

```
is
  x:integer;
  y:integer;
  part:string;
  found:boolean;
do
  --look for the next partName in the cycleTable
  part:=cycleTable[1,cycleTable.position];
  --look for the part in the stock of the warehouse
  stock.setCursor(1,1);
  found:=stock.find(part);
  if found then
    -- read coords from stock
    x:=stock["X",stock.cursorY];
    y:=stock["Y",stock.cursorY];
    --delete row in the stock-table
    stock.cutRow(stock.cursorY);
    -- increase position in cycle
    if cycleTable.position = cycleTable.YDim then
      cycleTable.position:=1;
    else
      cycleTable.position:=cycleTable.position+1;
    end;
    --take part put from warehouse
    warehouse.pe(x,y).cont.move(SingleProc);
  end;
end;
```

### 8.3.3 Virtual Warehousing

The "physical warehousing" (modeling of each MU) in the simulation can cause performance problems if you have a lot of parts in the system. All stored parts must be handled by Plant Simulation, which may slow down the simulation. Especially problematic are situations in which containers are used. These, in turn, may contain many individual parts. It can happen very quickly that millions of parts must be managed by Plant Simulation in the warehouse stock. It may be useful, in such cases, to register the containers when storing and destroy them. When the container is requested, then it and the contained parts are recreated and passed. The warehouse itself contains no parts or does so only briefly to generate the container and the parts located therein. During the registration of the containers, a range of information required for the new generation of containers and the parts contained therein is to be collected and stored.

- Class of the container
- Class of the part
- Name of the parts
- Number of parts in the container

For determining the classes of objects, use the SimTalk method `<object>.class`.

### Example: Virtual Warehouse

Plastic parts are manufactured and assembled. Each one of the six parts is mounted on a frame (Part 1 to Part 6 on MP). The six parts are manufactured on two machines (Part 1, Part 2 and Part 3 on Machine1; Part 4, Part 5, Part 6 on Machine2). After every 4,000 parts, the machines will be set up for a different part type. The parts are filled by the machines in containers with a capacity of 1,000 parts. The buffers are replenished if the buffer stock falls below 500 parts (request to the warehouse, then loading the container contents into the buffer). The mounting process takes 9.5 seconds. The warehouse has a safety stock of 10 containers with a capacity of 1,000 pieces per part. Create a frame that looks like Fig. 8.13.

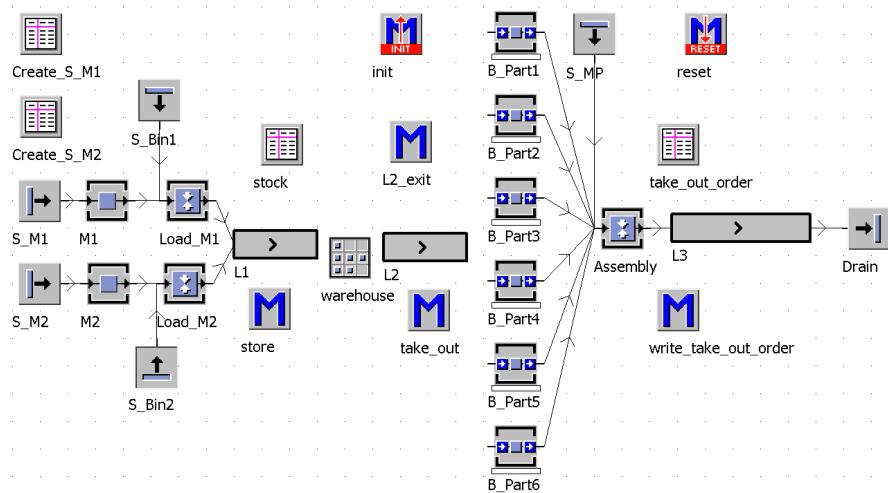


Fig. 8.13 Example Frame

Prepare the following settings: Sources: `S_M1` and `S_M2` blocking; interval: 2.4 seconds; sequence: cyclical, creation table (example `Create_S_M1` Fig. 8.14).

	object 1	integer 2
string	MU	Number
1	.MUs.Part1	4000
2	.MUs.Part2	4000
3	.MUs.Part3	4000
4		

**Fig. 8.14** Creation Table

The machines have a processing time of 2.4 seconds, a setup time of 30 minutes and an availability of 90 per cent with a MTTR of one hour. The assembly stations Load\_M1 and Load\_M2 have the following settings: processing time: zero seconds; assembly table: predecessors, 1,000 parts from the machines; assembly mode: attach MUs. L1 and L2 have a capacity of 1. With these lines, you can quite easily take into account the times for storage and retrieval. The warehouse has a capacity of six and is used only for creating the containers. The buffers upstream of the assembly have a capacity of 1,500 and no processing time. The assembly station loads one part of each buffer to a frame (which is produced by a source). Set the capacity of the bin to 1,000 and that of the frame to six. Design the parts in different colors. The table stock has a format as shown in Fig. 8.15.

	string 1	object 2	object 3	integer 4
string	Name	Bin	Part	Number
1				

**Fig. 8.15** Stock Table

The table take\_out\_job can be left in the original formatting.

Init method: The init method deletes the tables Stock and take\_out\_job, generates the initial inventory in the table stock (10 bins per 1,000 parts per type) and occupies the buffer upstream to assembly, each with 600 parts. The init method might look like this:

```

is
  i:integer;
do
  --initial stock
  for i:=1 to 10 loop
    stock.writeRow(1, stock.ydim+1,"Part1",
      .MUs.Bin,.MUs.Part1, 1000);
  next;
  --analogous complete for part2 to part6
  --initial stock buffer
  for i:=1 to 600 loop

```

```

    .MUs.Part1.create(B_Part1);
next;
--analogous complete for part2 to part6
end;

```

At the end of L1, the method `Store` is called (exit control front). The method writes a row in the stock table and destroys the bin along with the parts located therein.

```

is
do
  stock.writeRow(1,stock.Ydim+1,
    @.cont.name, @.class, @.cont.class, @.numMu);
  @.delete;
end;

```

The retrieval is performed by the generation of a bin and the parts contained therein. Since it may happen that multiple requests arrive at the same time, the retrieval requests are first entered into a table, which is then successively processed by the warehouse. In the present example, the following sequence is chosen: When the stock falls below the reorder point (500), a retrieval order is entered in the table `take_out_order` and the method `take_out` is called. If no retrieval job is processed (warehouse and L2 are empty) at present, the bin and the associated parts are created. At the end of L2 (exit control front), the bins are emptied into the appropriate buffer. After that, `take_out` is called again to execute any outstanding retrieval orders.

Method `write_take_out_order`: This method is called via the observer of the buffers (property `numMu`). The method calls once the method `write_take_out_order` at a stock of 500 parts (if the value has reduced):

```

(Attribut: string; oldValue: any)
is
do
  if ?.numMu < 500 and oldValue >=500 and
    ?.numMu>0 then
    --part name is part of the stations name
    take_out_order.writeRow(1,
      take_out_order.yDim+1,
      omit(?.name,0,3));
    take_out;
  end;
end;

```

Method `take_out`:

```

is
  part_name:string;
  found:boolean;

```

```

row:integer;
i:integer;
bin:object;
do
  --read first row from take_out_order
  -- then delete row
  part_name:=take_out_order[1,1];
  take_out_order.cutRow(1);
  --find part in stock
  stock.setCursor(1,1);
  found:=stock.find(part_name);
  if found then
    row:=stock.cursorY;
    --create bin in warehouse
    bin:=stock["Bin",row].create(warehouse);
    --create parts in the bin
    for i:=1 to stock["Number",row] loop
      stock["Part",row].create(bin);
    next;
    --delete row in stock
    stock.cutRow(row);
    --move bin to line
    waituntil L2.occupied=false prio 1;
    bin.move(L2);
  else
    messagebox("Warehouse run out of part: "+
      part_name,1,13);
    eventController.stop;
  end;
end;

```

At the end of the line, all parts of the bin are loaded into the appropriate buffer. Then, the bin is eliminated. If retrieval jobs still exist, then take\_out is called. Method L2\_exit (exit control front L2):

```

is
  destination:object;
do
  -- determine destination
  -- the destination consists of "B_"+partname
  --str_to_obj creates a object-link
  destination:=str_to_obj("B_"+@.cont.name);
  --load all parts into the buffer
  while @.occupied loop
    @.cont.move(destination);
    wait(0.1);
  end;
  --delete bin

```

```

@.delete;
--next take-out-order
if take_out_order.ydim > 0 then
    ref(take_out).methCall(0);
end;
end;

```

### 8.3.4 Extension of the Store Class

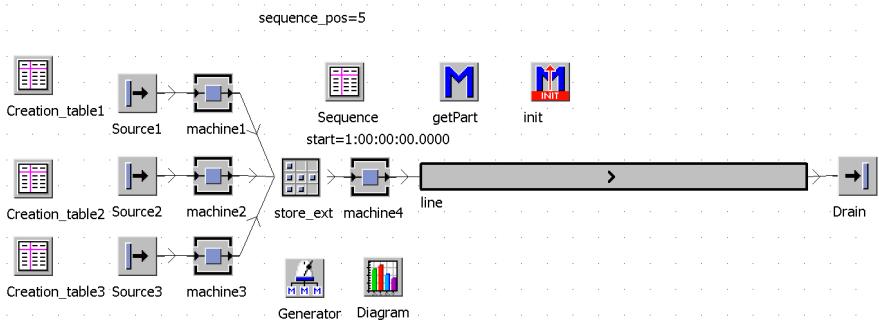
Plant Simulation provides a warehouse block (Store). You can connect the store using connectors with stations and store MUs. Retrievals must be realized in each case by SimTalk. Especially if you want to store and retrieve more than one part, the store lacks some features:

- Search for a free place + automatically register the stored goods in a stock list during storage
- Comfortable retrieval functions (e.g. by passing the MU-name)
- Parts-related stock statistics

With reasonable effort, the store class can be expanded using corresponding functions. The following example serves as a test box for the extended store class.

#### Example: Warehouse Simulation (Store Extension)

On each of three machines, two types of parts in a batch size of 100 parts are produced cyclically. The parts are stored in an interim storage facility and individually removed again in a fixed sequence. The courses of the stocks of the six different parts are to be recorded. Insert into the class library six parts (entities, T1-T6). Create a frame like in Fig. 8.16. Duplicate the store class in the class library and add an instance of the duplicate into the frame.



**Fig. 8.16** Example Frame

Prepare the following settings: The sources have an interval of four minutes; MU-selection: sequence cyclical; Source1: 100 T1, 100 T2; Source2: 100 T3, 100 T4; Source3: 100 T5, 100 T6; machine1, machine2, machine3: processing time: four minutes; setup time: one hour; availability: 75 per cent; two hours MTTR;

machine4: processing time: 2:10; line: speed 0.01 m/sec.; store\_ext: xDim: 100, yDim: 100.

### 8.3.4.1 Search a Free Place, Store, Update Stock List

A first step is to search automatically for a free place in the store. The parts must move to this place and an entry must be made in a stock list. For this purpose, we will develop two internal methods (user-defined attributes of type method): `getFreePlace` and `storePart`.

#### *Method getFreePlace*

The method must search the different areas of the store and check whether the place is free. The x and y coordinates of the place should be returned. For this purpose, the method is equipped with transfer parameters that are passed by reference (byRef). With a nested loop, the method `getFreePlace` might look like this:

```
byref x:integer;byref y:integer):Boolean
is
  i,k:integer;
do
  if self.~.full then
    return false;
  else
    for i:=1 to self.~.xDim loop
      for k:=1 to self.~.yDim loop
        if self.~.pe(i,k).cont=void then
          x:=i;
          y:=k;
          exitloop 2;
        end;
      next;
    next;
    return true;
  end;
end;
```

#### *Method storePart*

The storing must consist of several steps:

1. A free place must be sought.
2. The MU must be moved to this place.
3. The MU and the storage location (x, y) must be entered in a stock list.
4. The stock statistics must be updated.

Insert into the `store_ext` class the following user-defined attributes:

- `storePart` (Method)
- `stock` (table)
- `stockStatistics` (table)

Format the table stock like in Fig. 8.17.

	string 1	integer 2	integer 3	:
string	Part	X	Y	
1				

Fig. 8.17 Table Stock

The table StockStatistics needs to be formatted like in Fig. 8.18 Table StockStatistics.

	integer 1	integer 2	integer 3
string			
.			

Fig. 8.18 Table StockStatistics

The method storePart has the following content:

```
(part:object)
is
  x,y:integer;
  placeFree:boolean;
do
  -- look for a free place
  placeFree:=self.~.getFreePlace(x,y);
  if not placeFree then
    messagebox("Store "+self.~.name+" is full!",1,13);
  else
    -- move the part to the free place
    part.move(self.~.pe(x,y));
    --register
    self.~.stock.writeRow(1,self.~.stock.yDim+1,
      part.name,x,y);
    --actualize statistics
    --test, wheter the part is already a row index
    if self.~.stockPerPart.getRowNo(part.name) = -1
    then
      --create row index
      self.~.stockPerPart[0,
        self.~.stockPerPart.yDim+1]:=part.name;
    end;
    --increase stock
    self.~.stockPerPart[1,part.name]:=
      self.~.stockPerPart[1,part.name]+1;
  end;
end;
```

### 8.3.4.2 Search for and Retrieval of Parts

The following mechanism represents the retrieval process: After a waiting time of one day (variable start), the method `getPart` is called for the first time (method `init`). The method `getPart` calls the internal method `getPart` of the store and passes a part name according to a sequence. The method `getPart` of the store moves a corresponding part to `Machine4` (successor of the store). In the exit control of `machine4` (`rear`), a new part corresponding to the sequence is requested from the store.

#### Method `getPart` Sequence

The method `getPart` requests parts from the store in a particular sequence. The sequence is stored in the table `sequence`. It contains in the first column a series of part names (type: string)—e.g. `T1, T2, T3, T4, T5, T6`. Enter a sequence in the first column of the sequence table. The method `getPart` calls the method `getPart` of the store and passes the part name of the sequence table at the position `sequence_pos`. Thereafter, `sequence_pos` is incremented by one, and possibly set to one if the sequence is complete.

```
is
do
  --request part in the sequence on
  --position sequence_pos
  store_ext.getPart(sequence[1,sequence_pos]);
  sequence_pos:=sequence_pos+1;
  if sequence_pos > sequence.yDim then
    sequence_pos:=1;
  end;
end;
```

Enter the method `getPart` as exit control (`rear`) in `Machine4`. The first call is done by the method `init` after the distance start:

```
is
do
  wait(start);
  getPart;
end;
```

#### The method `store.getPart`

The method `getPart` finds the required part in the stock table and moves it to the successor of the store. The related row in the stock table is deleted. The number of parts in the table `stockPerPart` is decreased by one. If the part does not exist, an error message is shown.

```
(part:string)
is
  found:boolean;
  successor:object;
```

```

x,y:integer;
do
  --find the part in stock
  successor:=self.~.succ;
  waituntil successor.operational and successor.empty
    prio 1;
  self.~.stock.setCursor(1,1);
  found:=self.~.stock.find(part);
  if found then
    x:=self.~.stock[2,self.~.stock.cursorY];
    y:=self.~.stock[3,self.~.stock.cursorY];
    self.~.pe(x,y).cont.move(successor);
    self.~.stock.cutRow(self.~.stock.cursorY);
    --reduce stock of this part
    self.~.stockPerPart[1,part]:=-
      self.~.stockPerPart[1,part]-1;
  else
    messagebox("Part "+ part+
      " not in store "+self.~.name+" available!",1,13);
  end;
end;

```

### 8.3.4.3 Stock Statistics

To control the simulation, it is important to visualize the course of the stock of the individual parts. These stock levels must be written in a table periodically. The table can be easily visualized by a chart block. Insert for statistical purposes in the class store\_ext as user-defined attributes a method writeStockStatistics, a method init and a table stockStatistics. Assign to the first 10 columns of the table stockStatistics, the data type integer and activate the column indexes. The method "writeStockStatistics" writes the data located in rows in the table stockPerPart side by side in a new row at every call of the method. To remain flexible, even the column index is reset:

```

is
  i:integer;
  rowNumber:integer;
do
  --write row index of StockPerPart in the column
  --index of StockStatistics
  for i:=1 to self.~.StockPerPart.yDim loop
    self.~.stockStatistics[i,0]:=-
      self.~.stockPerPart[0,i];
  next;
  rowNumber:=self.~.stockStatistics.yDim+1;
  --insert values
  for i:=1 to self.~.stockPerPart.yDim loop

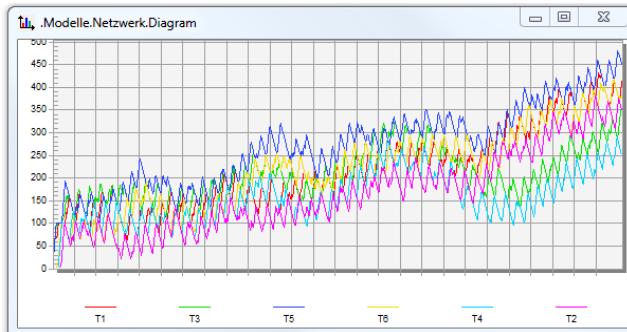
```

```

    self.~.stockStatistics[i,rowCount]:= 
        self.~.stockPerPart[1,i];
    next;
end;

```

The method writeStockStatistics is called every hour by a generator. If you display the table stockStatistics, you get a visualization of the inventory history of all parts (Fig. 8.19).



**Fig. 8.19** Stock Levels

### Store\_ext.init

In the init method of the store, the Stock and Statistics tables must be deleted.

```

is
  i:integer;
do
  --delete stock
  self.~.stock.delete;
  --delete all row indexes of stockPerPart
  for i:=1 to self.~.stockPerPart.yDim loop
    self.~.stockPerPart[0,i]:="";
  next;
  self.~.stockPerPart.delete;
  --delete all column names of stockStatistics
  for i:=1 to self.~.stockStatistics.xDim loop
    self.~.stockStatistics[i,0]:="";
  next;
  --delete stockStatistics
  self.~.stockStatistics.delete;
end;

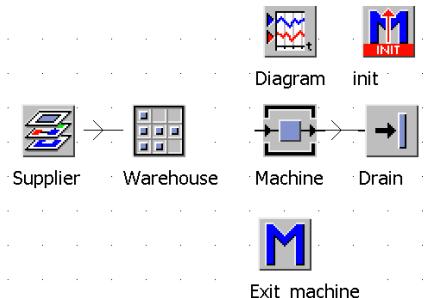
```

### 8.3.5 Simplified Warehousing Model

In the following, a simplified warehousing model should be created. The main limitations are:

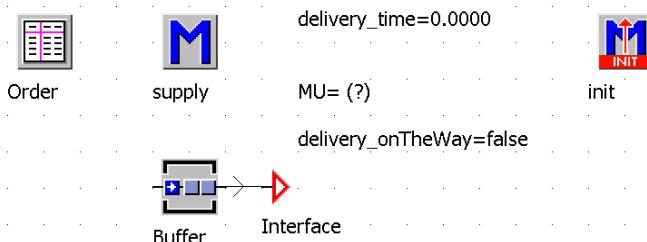
- only one part, only one supplier, constant delivery time
- constant consumption

Warehousing models must always take into account consumption and supply (replenishment). The simplified warehousing model is intended to represent the following: A SingleProc produces parts at a constant interval. The blanks for it are to be removed from a warehouse. After falling below a specified order quantity, blanks are ordered from the supplier, which delivers them after a specified delivery time and stores them in the warehouse. The delivery process of the supplier is not investigated. Create a frame like in Fig. 8.20.



**Fig. 8.20** Example Frame

The supplier frame must contain elements corresponding to Fig. 8.21.



**Fig. 8.21** Sub-frame Supplier

Prepare the following settings:

- Machine: processing time: two minutes
- Warehouse capacity: 1,500 parts
- Supplier.delivery\_time: one day (date type: time)
- Supplier.MU: link to an entity (e.g. .MUs.Entity)
- Supplier.delivery\_onTheWay (data type: Boolean; start value = false)

For the warehouse, we need some attributes to map an ideal-typical behavior.

### Reorder Point

The reorder point (or reorder level) is the stock below which an order is triggered. The triggering of the order must be early enough so that the stock not fall below the safety stock during the replenishment lead time.

### Safety Stock

The safety stock is intended to ensure the supply of the production in case a delivery arrives late.

Calculation:

$$\text{Reorder point} = \text{Consumption per time-unit} * \text{Replenishment lead time} + \text{Safety stock}$$

$$\text{Safety stock} = \text{Consumption per time-unit} * \text{Replenishment lead time}$$

$$\text{Reorder point} = 2 * \text{Consumption per time-unit} * \text{Replenishment lead time}$$

### Maximum Stock Level

The maximum stock level indicates the amount of material that may be present to the maximum level in stock. With its help, an excessive stock and, thus, an unnecessarily high capital lookup should be avoided in the warehouse.

### Order Quantity

With each order, the warehouse is filled to the maximum stock level in the ideal case. It must be considered that parts are consumed out of the warehouse during the replenishment lead time. The order quantity, thus, results from the difference between the reorder point and maximum stock level plus the consumed parts during the delivery period:

$$\text{Order quantity} = \text{Maximum stock level} - \text{Reorder point} + (\text{Consumption per time-unit} * \text{replenishment lead time})$$

All stocks can be represented as in Fig. 8.22:

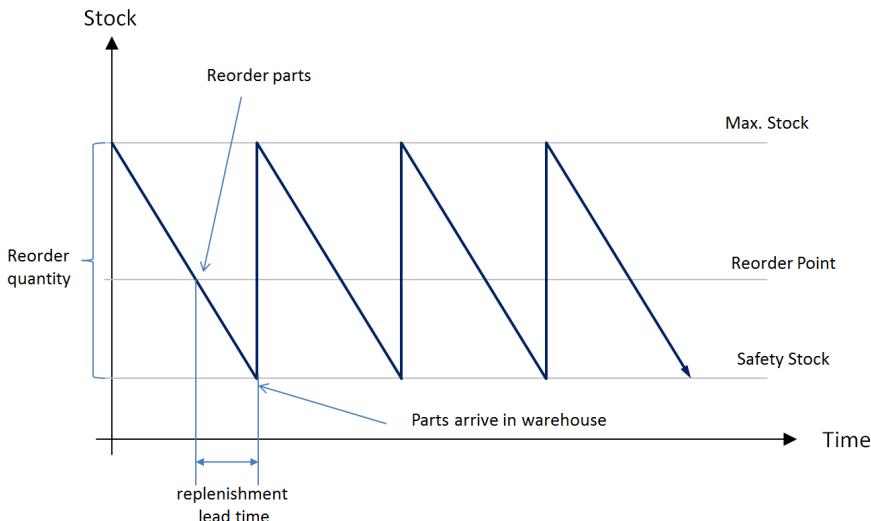


Fig. 8.22 Warehousing Model

The stock decreases constantly during the observation period and reaches the reorder point at which the order of new material is released. On reaching the safety stock (after the replenishment lead time), the ordered material arrives (ideally).

For the material disposition, we need some attributes in the warehouse block. Create them according to Fig. 8.23.

Name	Value	Type	C
checkStock		method	...
maximumStock	1500	integer	...
reorderPoint	0	integer	...
safetyStock	0	integer	...

Fig. 8.23 User-defined Attributes Warehouse

### Modeling Approach: Material Consumption

The warehouse is filled at the beginning of the simulation (init method) to the maximum stock with parts. The first part is moved to the machine. The method exit\_machine (exit control of the machine, rear) moves the next part from the warehouse to the machine. In this way, the consumption of parts is taken into account. Unfortunately, as the warehouse has no exit control, we must call the method for inventory tracking (checkStock) even within the method exit\_machine.

Part 1: Fill warehouse, move the first part to the machine; use method init of the main frame

```
is
do
  --fill warehouse
  while warehouse.numMu < warehouse.maximumStock loop
    .MUs.part.create(warehouse);
  end;
  --move the first part
  warehouse.cont.move(machine);
end;
```

The method exit\_machine should have the following content:

```
is
do
  waituntil warehouse.occupied prio 1;
  --move one part to the machine
  warehouse.cont.move(?);
  --call inventory management of the warehouse
  warehouse.checkStock;
end;
```

The machine processes all parts of the warehouse, after which the simulation is terminated.

### Trigger Orders

After each removal from the warehouse, the stock (numMU) must be checked. If this is less than the reorder point, an order in the supplier sub-frame is triggered (stored in the table).

For later experiments, the reorder point is to be recalculated in the init method at the beginning of each experiment run. Later, you can dynamically perform experiments with different values. The reorder point can be calculated by dividing the time of delivery of the supplier and the processing time of the machine and addition of the safety stock. Set the safety stock to 100 parts. Extend the init method:

```
--calculate the reorder point from basis data
warehouse.reorderPoint:=
warehouse.safetyStock+supplier.delivery_time
/machine.procTime;
```

You need to format the table supplier.order as shown in Fig. 8.24.

	string 1	integer 2
string	Part	Quantity
1		

**Fig. 8.24** Table Order

The method checkStock should contain the following program:

```
is
do
  --check stock < reorder point
  if self.~.numMu < self.~.reorderPoint and
    supplier.delivery_onTheWay=false then
      -- register order
      supplier.order.writeRow(1,
        supplier.order.yDim+1,
        "Part",
        self.~.maximumStock-self.~.reorderPoint +
        supplier.delivery_time/machine.procTime);
      -- lock for the next calls
      supplier.delivery_onTheWay:=true;
    end;
end;
```

If you run the simulation, an order is entered into the table order.

### Delivery

When an order for the supplier exists, the parts have to arrive at the warehouse after the delivery time. There are a number of ways to realize this. In our simple

example, the transport of parts to the warehouse should not be taken into account. After expiry of the delivery time, the ordered parts are created in the buffer and passed via the connector to the warehouse. When the buffer is empty again, the supplier is released for a new order. Thus, the method supply should have the following content:

```

is
  i:integer;
  quantity:integer;
do
  waituntil order.yDim > 0 prio 1;
  -- there is an order
  -- wait the delivery-time
  wait(delivery_time);
  -- the second column of the first row contains
  --the quantity
  quantity:=order[2,1];
  --delete order
  order.cutRow(1);
  --create parts in the buffer
  for i:=1 to quantity loop
    MU.create(buffer);
  next;
  --wait, until the buffer is empty again,
  --than release
  waituntil buffer.empty prio 1;
  delivery_onTheWay:=false;
  --recursion, wait for the next order
  self.methCall(0);
end;

```

In the init method of the supplier, entries may need to be deleted from the order table and the method Supply must be called once.

```

is
do
  order.delete;
  supply;
end;

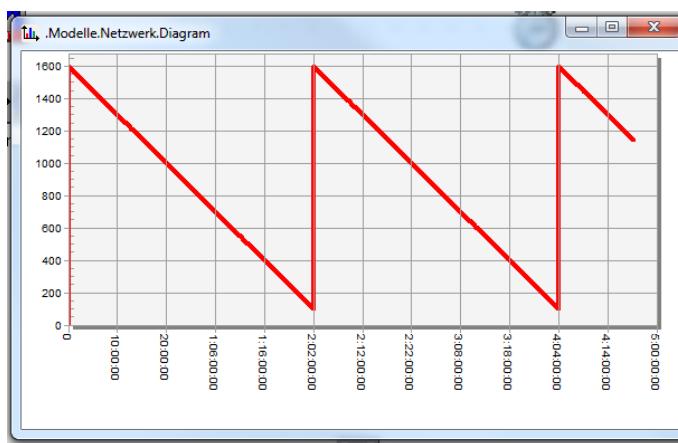
```

Now, the simulation runs.

A chart block is intended to visualize the stock of the warehouse. For this, the following settings are required:

- Data source: Input Channels → warehouse.numMu
- Mode: Plot
- Display: category—Plotter
- Axes: Number of values = 20,000
- Axes: Range X = 5:00:00:00

The plotter shows the theoretically expected course (Fig. 8.25).



**Fig. 8.25** Warehouse stock

### 8.3.6 *Warehouse Key Figures*

To control and evaluate warehousing, there is a whole series of key figures that can be integrated into the simulation. The calculation of some of these figures causes some additional effort in the modeling. Table 8.4 shows a small selection of possible key figures.

**Table 8.4** Warehousing Key Figures

Name	Calculation
Days of inventory	Average inventory/Average consumption per time unit
Safety factor	(Safety stock/Average inventory) * 100 percent
Stock turnover rate	Total consumption/Average consumption
Stock turnover period	360 days/Stock turnover rate

#### Average Inventory/Stock

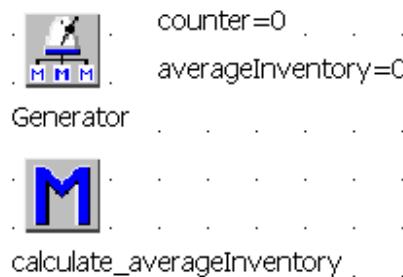
For most of the key figures, we need the average inventory (within a certain time period). For the calculation of the average inventory, there are three options:

- Continuous calculation of the average
- Periodic recording of the stock (e.g. at intervals of one hour) and calculating the average interval based on the recorded values
- Calculation of the average inventory based on statRelativeOccupation

To perform a) for an ongoing inventory calculation we need the following elements:

- A generator for the periodic call of the calculation
- A method for calculation (calculate\_averageInventory)
- A variable for counting the number of calculations (counter; data type: integer; initial value = 0)
- A variable for the average inventory (averageInventory; data type: real; initial value = 0)

Set up the elements from Fig. 8.26 in the frame.



**Fig. 8.26** Average Inventory

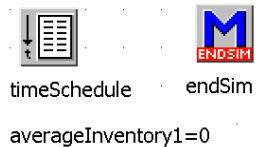
The generator calls the method calculate\_averageInventory (distance control) at regular intervals (10 minutes). The method multiplies the current average with the value of the counter and adds to the product the current value. Then the counter is incremented by one, and the value is divided by the counter. The new average is stored in the variable averageInventory. The method calculate\_averageInventory has the following content:

```

is
  val:real;
do
  val:=averageInventory*counter;
  counter:=counter+1;
  val:=val+warehouse.numMu;
  averageInventory:=val/counter;
end;
  
```

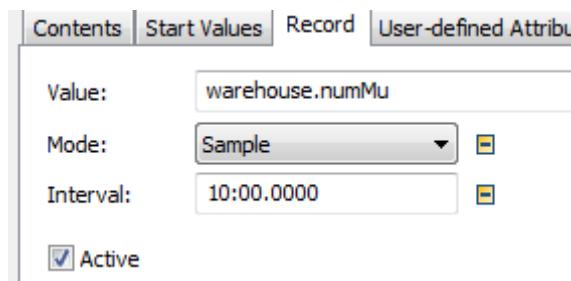
The longer the distance of the generator, the less accurate is the calculated value. If the distance you choose is too small, then the simulation performance is adversely affected.

To perform b) for periodic records, you can use a TimeSequence. The timeline has a method `meanValue`, with which you can calculate the average inventory. Insert the elements from Fig. 8.27 into the frame.



**Fig. 8.27** Average Inventory (timeSchedule)

Change the second column of the TimeSchedule to data type integer. The settings in the tab Record should look like in Fig. 8.28.



**Fig. 8.28** TimeSchedule

The method `endSim` is automatically called at the end of the simulation by the `EventController` (For this, you need to define an end in the `EventController`). The average inventory then can be calculated as follows:

```
is
do
  averageInventory:=
    timeSchedule.meanValue({2,1}..{2,*});
end;
```

To perform c) only one line of SimTalk is necessary:

```
is
do
  averageInventory2:=
    warehouse.statRelativeOccupation*warehouse.capacity;
end;
```

## Warehouse Key Figures

The warehouse key figures are to be calculated. Add a table (warehouse\_key\_figures) according to Fig. 8.29 into the frame.

	string 0	real 1
string	Key figure	Part
1	Days of inventory	
2	Safety factor	
3	Stock turnover rate	
4	Stock turnover period	

**Fig. 8.29** TableFile Warehouse Key Figures

Note: The average consumption can be calculated using the statistics of the machine. You must relate the consumption to one time unit (e.g. days) in order to get the "days of inventory" in days. The stock turnover is also based on a period (e.g. 168 hours for a month or 210 days for a year).

Enter 210 days as the end of the EventController; we will relate the stock turnover to one year with 210 working days. The extensions of the endSim method for calculating the warehouse key figures could look like the following:

```

is
  av_consumption :real;
do
  averageInventory1:-
    timeSchedule.meanValue({2,1}..{2,*});
    --calculate days of inventory
    --average consumption = consumption/duration (days)
  av_consumption:=machine.statNumIn/
    (EventController.simTime/86400);
  warehouse_key_figures[1,1]:=
    averageInventory/av_consumption;
    --safety factor
  warehouse_key_figures[1,2]:=
    warehouse.safetyStock/averageInventory;
    --stock turnover rate
  warehouse_key_figures[1,3]:=
    machine.statNumIn/averageInventory;
    --stock turnover period
  warehouse_key_figures[1,4]:=
    360/warehouse_key_figures[1,3];
end;

```

### 8.3.7 Storage Costs

For some studies, it is necessary to integrate cost information into the simulation (e.g. optimal order quantity).

#### Warehouse Costs

All costs incurred in the warehouse are recorded as warehouse costs (excluding interest costs). These are mainly:

- Occupancy costs
- Labor costs
- Rent
- Amortization, maintenance
- Heating, lighting
- Material transfer, insurance
- Loss, deterioration

The warehouse costs are calculated for a period (usually per month). If you set the warehouse costs in relationship to the monetarily valued average inventory, you get the warehouse cost rate. It indicates how much to charge related to the stock value to cover the warehouse costs (percentage surcharge). It could be calculated as follows:

$$\text{Warehouse cost rate} = \text{Warehouse costs} / \text{valued average inventory} * 100\%.$$

The average inventory is normally valued by the production costs.

#### Interest Costs

The average fixed capital in the warehouse must be remunerated. Servicing the interest on an imputed interest rate is taken into account.

$$\text{Imputed interest} = \text{Valued average inventory} * \text{Imputed interest rate} / 100$$

#### Storage Cost Rate

The storage cost rate includes as a surcharge on the value of the average inventory all costs that are related to the warehousing. The warehousing cost rate per value unit per time unit is given by:

$$\text{Storage cost rate} = \text{Warehouse cost rate} + \text{Imputed interest rate}$$

#### Storage Costs

The storage costs can be calculated alternatively:

$$\text{Storage costs} = \text{Warehouse costs} + \text{interest costs}$$

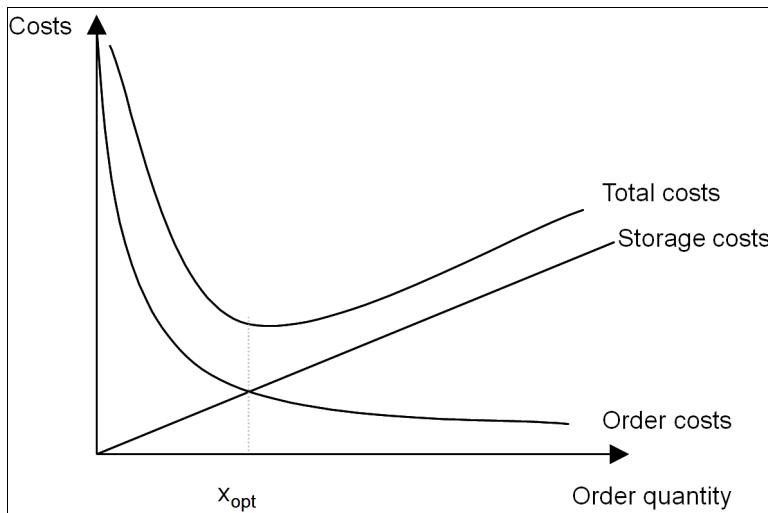
Or

$$\text{Storage costs} = \text{Valued average inventory} * \text{Storage cost rate}$$

The storage cost rate is usually a fixed value for modeling. As such, it can simply be stored as an attribute in the warehouse. The average inventory is variable and regularly the subject of investigations.

### 8.3.8 Economic Order Quantity

When optimizing the procurement quantities the following problem is to be solved: With increasing quantities, the order costs are reduced (less orders); simultaneously storage costs rise due the higher average inventory (see Fig. 8.30).



**Fig. 8.30** Economic Order Quantity

### Simulative Determination of the Economic Order Quantity

The total sum of order costs and storage costs can also be determined through simulation. By varying the order quantity (or the maximum inventory), we can determine a minimum of total costs. Experimental setup: Integrate the following variables into the simulation (Fig. 8.31).

```

economic_order_quantity
storage_cost_rate=45
cost_order=35
number_orders=0
part_value=30

storage_costs=0
order_costs=0
total_costs=0

```

**Fig. 8.31** Variables Economic Order Quantity

All variables are of the data type real. The cost calculations will be carried out in the `endSim` method at the end of each simulation run. For this purpose, the following additions are required:

```

--calculate the number of orders
number_orders:=(warehouse.statNumIn-
warehouse.safetyStock) /

```

```

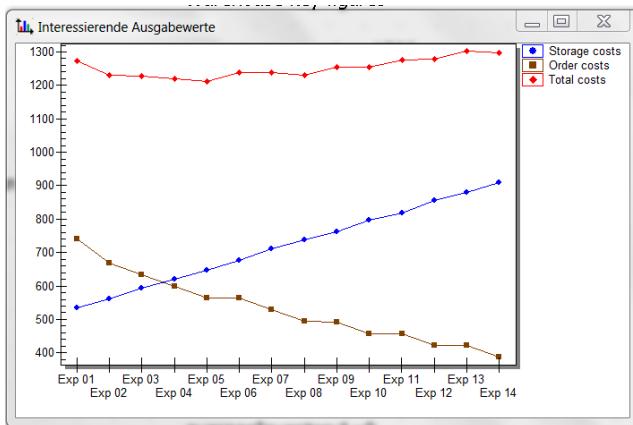
warehouse.maximumStock-warehouse.safetyStock);
order_costs:=cost_order*number_orders;
--storage_cost_rate for one year in %
-- simulation duration: one month
storage_costs:=
averageInventory*part_value*storage_cost_rate/100/12;
total_costs:=storage_costs+order_costs;

```

The following settings are required in the ExperimentManager to simulate the economic order quantity:

- Output values: Order\_costs, Storage\_costs, Total\_costs
- Input values: warehouse.maximumStock
- Experiments: maximum stock from, e.g. 900 to 1,600 in increments of 50

Set in the EventController an end of 21 days (to simulate one month). After running the experiments, you should get a diagram like in Fig. 8.32.



**Fig. 8.32** Diagram for Economic Order Size

The economic order size is near 950 (maximum stock 1,050 – safety stock 100).

### 8.3.9 Cumulative Quantities

Cumulative quantities in warehousing visualize cumulative additions and disposals of the warehouse over a time axis. Both values are provided by Plant Simulation:

- statNumOut (cumulative disposals)
- statNumIn (cumulative additions)

A chart block is sufficient for representing this. Create the following settings:

- Data source: Entrance Channels (Fig. 8.33)

string 0	string 1
string	Quantity
1 cum. additions	warehouse.statNumIn
2 cum. disposals	warehouse.statNumOut

Fig. 8.33 Setting Plotter

- Mode: sample
- Interval: 10 minutes
- Display: category plotter—line

Set in tab Axis, a sufficiently high number of values and a corresponding range of values for the x-axis. The result should look like in Fig. 8.34:

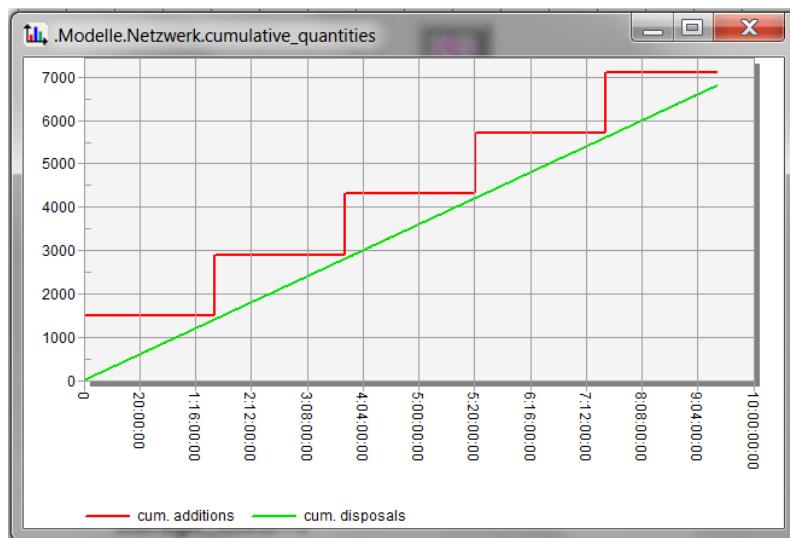


Fig. 8.34 Cumulative Quantities

On the basis of the cumulative quantities, you can easily determine problems in the design of delivery and consumption (both lines should run "parallel").

## 8.4 Procurement

To model the incoming material flow of a company, the warehouse objects must be extended by the procurement functions. This concerns the activations of orders for the supplier and the monitoring of orders that are already underway (which have to be taken into account in the calculation of orders). With the help of the generated orders, you can attach the deliveries from the suppliers including the corporate interfaces (inbound) to the simulation.

### **8.4.1 Warehousing Strategies**

Warehousing strategies (or stock strategies) are used to make decisions about the time and quantity of procurements and deliveries of materials. With respect to time, two cases are possible:

- The time of the order is dependent on the stock (e.g. it is only ordered when the reorder point is reached or an order is created after each stock removal)
- The warehouse will be filled up to a certain stock level at regular intervals

Similarly, there are different options in terms of quantity:

- A fixed quantity is ordered
- The stock is refilled to a certain stock (depending on the consumption, the required quantity is different)

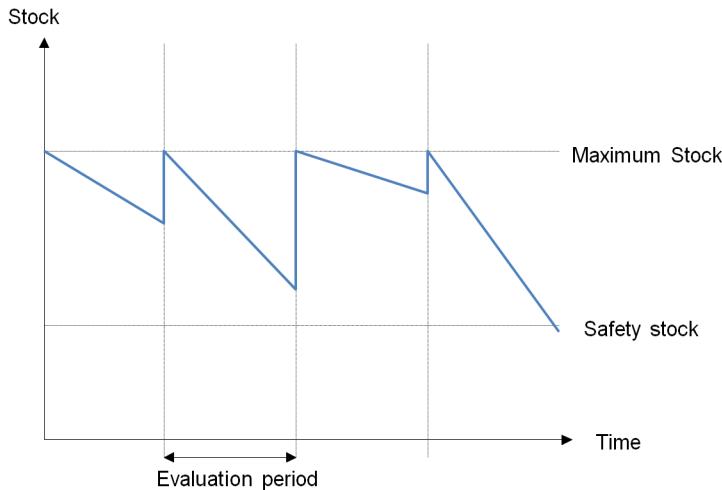
### **8.4.2 Consumption-Based Inventory Replenishment**

Consumption-based inventory replenishment is mainly used for materials and supplies with regular consumption or for relatively low-value materials. Common in practice is e.g. the replenishment of the inventory to a basic stock (often with technical specifications), such as in silos, bunkers, tanks, etc. The amount required to replenish stocks will be ordered on the review date depending on the stock and the expected consumption during the delivery period. The consumption-based inventory replenishment is divided into:

- Order rhythm method
- Reorder point method

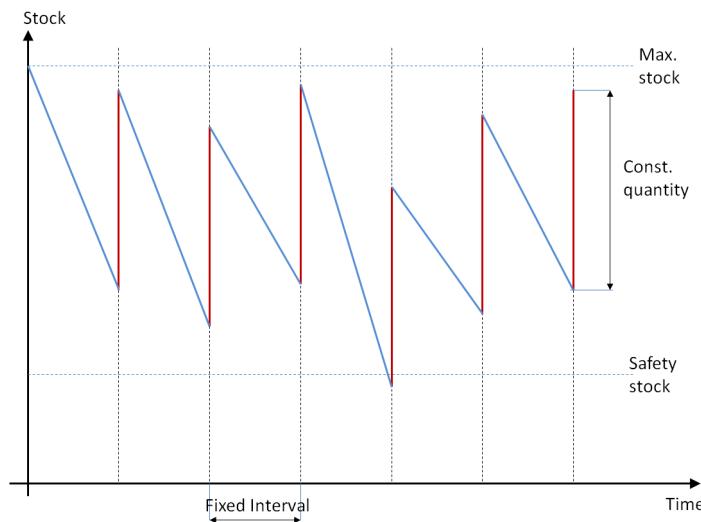
#### **8.4.2.1 Order Rhythm Method**

The order rhythm method is characterized by a fixed procurement cycle that depends especially on consumption between the review times. A prerequisite for the process is a periodic stock check, which should determine the consumption of the previous period. The more often that checks between the order cycles are performed, the more accurately can the order quantity be specified during procurement. This type of inventory replenishment works satisfactorily at only relatively constant demand and/or with relatively high-safety stocks (Fig. 8.35).



**Fig. 8.35** Order Method

The modeling will be shown in a special case: There are materials (e.g. bulk material, gas, fluids) that are delivered over tour plans at regular intervals in constant amounts (e.g. silo filling). Here, the supplier must be notified about how often the delivery must take place (Fig. 8.36). In case of changes in the production, this interval must be adjusted.



**Fig. 8.36** Order Rhythm with Constant Quantities

The following example should serve as a frame for modeling such cases. A company manufactures plastic parts on two different injection molding machines.

The supply of the machine takes place via a pipe system from a central silo. The silo is filled at regular intervals. With the simulation, the calculation of the filling should be tested. Two different parts are manufactured on each machine. After a specified number of satisfactory parts, the machine will be set up for the next part type. Integrated into the machines are quality inspection facilities. Rejects are crushed and fed into the central silo again. Data:

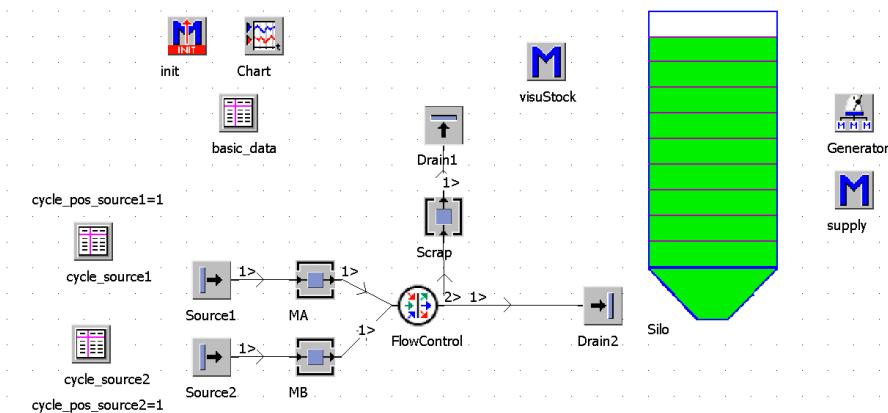
**Table 8.5** Data Example

Part	Setup after x parts	Granules consumption	Reject quote	Cycle time
A	10,000	150 g	10%	2 sec.
B	10,000	200 g	10%	1.5 sec.
C	5,000	120 g	3%	3 sec.
D	30,000	20 g	15%	15 sec.

Machine MA manufactures parts A and B; machine MB manufactures C and D. The central silo has a capacity of 45 tons. The filling takes place at a fixed rate of 20 tons. The machines have a setup time of three hours. For the simulation of the consumption of granules, the amount in the pipes will be neglected.

### Modeling Approach

In the model, the material consumption and the cyclical deliveries are especially of interest. The stock of the silo can be represented logically (as the value of a user-defined attribute). For better visualization, you can animate the icon of the silo (e.g. according to the filling ratio). Create a frame like in Fig. 8.37.



**Fig. 8.37** Example Frame

A SingleProc serves as an injection-molding machine. Before the part is transferred to the machine, the corresponding processing time is set. In the exit control, the material consumption is "booked" in the silo and an attribute (ok) of

the part is set to true or false corresponding to the reject rate. If the part is in order (ok = true), then the part is registered. If a batch is entirely processed, the cycle of source is switched. A FlowControl is used to separate the okay from the not-okay parts. The weights of the not-okay parts are added in the exit control of the station Scrap to the silo inventory again. The periodic deliveries are easiest to realize using a generator and a method (distance control).

### Silo

Insert into the block for the silo two user-defined attributes: inventory (real) and maxInventory (real, value 40,000).

### Basic Data

Create the table Basic\_data like in Fig. 8.38.

	string 0	integer 1	real 2	real 3	time 4
string	Part	Setup after x parts	granules consumptio	reject rate	cycle time
1	A	100000	150.00	0.10	2.0000
2	B	100000	200.00	0.10	1.5000
3	C	50000	120.00	0.03	3.0000
4	D	30000	20.00	0.15	2.0000

Fig. 8.38 Basic Data

Assign information to the sources about the number and types of parts. This will be done through two cycle tables. Format the cycle table according to Fig. 8.39 (example of cycle\_source1).

	string 1	integer 2	integer 3
string	Part	Target quantity	Actual quantity
1	A	100000	0
2	B	100000	0
3			

Fig. 8.39 Cycle Table

Set for each machine a setup time of three hours.

### Part Creation

First, create a part in the class library (duplicate the entity). Insert into the part two user-defined attributes: okay (Boolean, value: true) and materialConsumption (real). Adjust both sources so that the new part is produced. The production figures refer okay values; therefore, the source does not "know" for sure how many parts to produce. Instead, according to the audit, the okay parts are counted. If the required number is reached, retooling is initiated. To do this, follow the steps:

1. For the setup, it is sufficient to have MUs with different names. In the first step (exit control source front), the source determines the current part from the cycle\_table and the value of the variable cycle\_pos\_source.
2. The MU is named accordingly and the processing time of the machine is set if it is empty.
3. The part is moved to the machine.
4. The exit control of the machine counts the okay parts and sets, when the batch is processed, the variable cycle\_pos\_source to the next cycle. In addition, the exit control marks a fixed number of parts as scrap.

You can use the source to produce parts when you need them. The setting for this is blocking and the interval is 0. The source then permanently provides a part. Create in both sources exit controls (front). The exit control of Source1 looks like the following:

```
is
do
  --name part according to cycle_table
  @.name:=cycle_source1[1,cycle_pos_source1];
  --set material consumption
  @.materialConsumption:=basic_data[2,@.name];
  --set procTime of the succeeding station
  self.~.succ.procTime:=basic_data[4,@.name];
  @.move;
end;
```

Create an analogous exit control for Source2. The exit control of the machine MA has the following content:

```
is
  scrap:real;
  value:real;
do
  --register material consumption
  silo.inventory:=silo.inventory-
    (@.materialConsumption/1000);
  --scrap??
  scrap:=basic_data[3,@.name];
  value:=z_uniform(23,0,1);
  if value<=scrap then
    --scrap --> will be sorted by the FlowControl
    @.okay:=false;
  else
    --count
    cycle_source1[3,cycle_pos_source1]:=cycle_source1[3,cycle_pos_source1]+1;
```

```

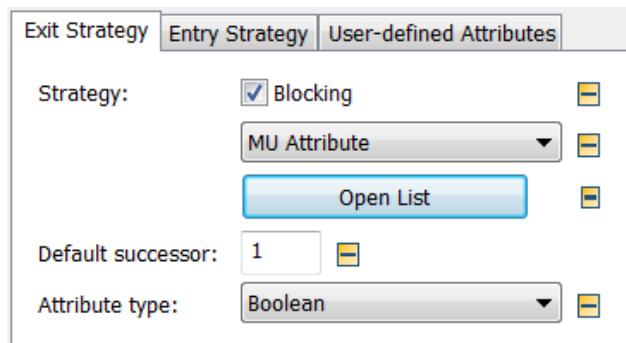
--finished?
if cycle_source1[3,cycle_pos_source1]>=
  cycle_source1[2,cycle_pos_source1]-1 then
  cycle_source1[3,cycle_pos_source1]:=0;
  --next position in the table
  cycle_pos_source1:=cycle_pos_source1+1;
  if cycle_pos_source1 > cycle_source1.yDim then
    cycle_pos_source1:=1;
  end;
end;
@.move;
end;

```

Create for MB a similar exit control (change the object links).

### Setting of FlowControl, Quality Test

The FlowControl has to distribute the MUs according to the value of the property "okay." For this, the settings in Fig. 8.40 in the tab Exit Strategy are necessary.



**Fig. 8.40** Setting FlowControl

The list contains entries like in Fig. 8.41 (Successor2 denotes the scrap branch).

	Attribute	Value	Successor
1	okay	true	1
2		false	2

**Fig. 8.41** Successor List

### Scrap/Rejects, Refluxing into the Silo

At the end of the station Scrap, the weights of the parts are added to the inventory of the silo (exit control rear).

```
is
do
  --back into the silo
  silo.inventory:=
    silo.inventory+(@.materialConsumption/1000);
end;
```

### Delivery

The delivery is done by the regular call of the method Supply (generator interval control). The method adds the delivery quantity to the stock of the silo.

```
is
do
  silo.bestand:=silo.bestand+20000;
end;
```

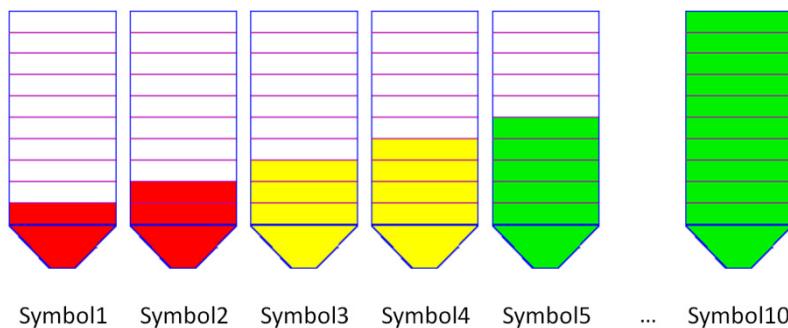
### Initializing

The following initial settings are necessary:

```
is
do
  --initialize inventory
  silo.inventory:=silo.maxInventory;
  --delete cycle tables
  cycle_source1.delete;
  cycle_source2.delete;
  --cycle to start
  cycle_pos_source1:=1;
  cycle_pos_source2:=1;
  --distribute parts and output to the cycle_tables
  cycle_source1[1,1]:=basic_data[0,1];
  cycle_source1[2,1]:=basic_data[1,1];
  cycle_source1[1,2]:=basic_data[0,2];
  cycle_source1[2,2]:=basic_data[1,2];
  cycle_source2[1,1]:=basic_data[0,3];
  cycle_source2[2,1]:=basic_data[1,3];
  cycle_source2[1,2]:=basic_data[0,4];
  cycle_source2[2,2]:=basic_data[1,4];
end;
```

### Animation of the Filling Level of the Silo

Using different symbols, you can show the level of the silo in the simulation. To do this, add 10 icons (numbers one to 10, one for 10 per cent level and 10 for 100 per cent) for the silo (e.g. like in Fig. 8.42).

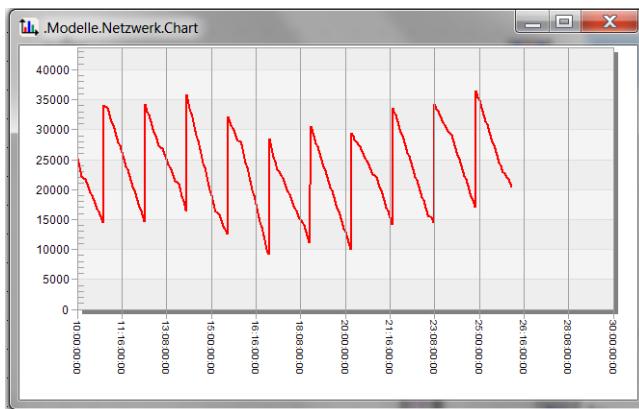


**Fig. 8.42** Silo Icons

The icons can be updated via a method that is called by an observer of the silo (monitor the attribute inventory). The method visuStock could look like the following:

```
(attr:string;altValue:any)
is
  symbolNo:integer;
do
  symbolNo:=
    ceil((silo.inventory/silo.maxInventory)*10);
  if symbolNo /= silo.curriconNo then
    silo.curriconNo:=symbolNo;
  end;
  silo.redraw;
  --silo is empty --> stop
  if silo.inventory < 0 then
    messageBox("Silo empty",1,1);
    eventController.stop;
  end;
end;
```

To illustrate the silo inventory over time, a plotter is suitable (Chart—tab Display—Plotter; see Fig. 8.43).



**Fig. 8.43** Silo Stock

#### 8.4.2.2 Reorder Point Method

The reorder point is the amount of available stock on which an order is triggered. Therefore, initially, you have to decide the order quantity and the point at which an order process is triggered. For this purpose, each booking will be checked after storage retrieval, regardless of whether the reorder point is reached or not. The reorder point method is practiced in two ways:

- immediate stock replenishment
- long-term stock replenishment

##### Immediate Stock Replenishment

Immediate stock replenishment is applied to materials whose replenishment can be made between two stock decreases, because their procurement times are short. The order point is calculated as follows:

$$\text{Order Point} = \text{Replenishment lead time} * \text{Consumption rate} + \text{Safety stock}$$

The replenishment lead time mainly includes order placement, order processing, order taking, order processing, transportation and material input.

##### Long-term Stock Replenishment

In long-term stock replenishment, it is assumed that further retrievals are made from the stock between the order of the material and the arrival of the material. Hence, inventory management must also consider the stocks that are already ordered. The order process is then carried out in the following steps:

1. With an inventory check, determine whether the stock including the previously ordered quantity reaches the reorder point.
2. If the reorder level is reached or exceeded, the amount must be ordered, that fills up the stock up to the basic stock taking into account the existing and ordered quantities.

If the inventory check returns that the available and ordered amount is greater than the reorder point, then no order is initiated.

### Example: Multi-container System

One particular form of the reorder point method is the multi-container system. In this case, a specific material is provided with at least two containers. When a container is empty, it automatically results in reordering. During the delivery time, the consumption is carried out from one or more backup containers. To demonstrate, we will model a small section from a production line.

In a production line, adhesives are applied at two stations. The adhesive is provided in two tanks at the production station. When the adhesive is consumed in a tank, then the adhesive is pumped from the second tank into the first tank. This pumping takes an hour; during this time, no machining is possible. The filling of the second tank can then be done parallel to the adhesive application and is considered in the delivery time (delivery time of supplier + an hour for filling). Two products are manufactured with a random distribution and with different consumption amounts of adhesive. Create a frame according to Fig. 8.44.

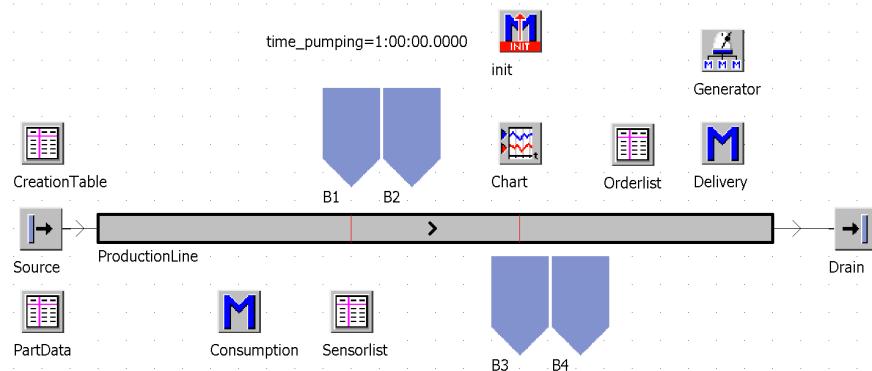
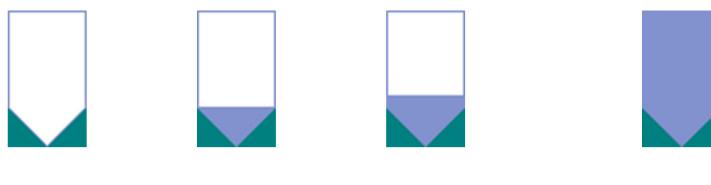


Fig. 8.44 Example Frame

Create for the containers a set of symbols like in Fig. 8.45.



Symbol 0    Symbol 1    Symbol 2    ...    Symbol 10

Fig. 8.45 Icon Set for the Container

### Tank Simulation

The filling level of the tank should be simulated only as the value of a user-defined attribute. The animation will then show in 11 steps, as the filled level moves from empty to full. In addition, a reference to the reserve tank is to be

stored in the main tank. The animation of the symbol will be designed as an internal method of the tank. Duplicate a buffer in the class library. Insert into the buffer a set of user-defined attributes according to Fig. 8.46.

Name	Value	Type	C.	I.	3I
fillingLevel	0	real	*		
maxFilling	250	real	*		
reserveTank	(?)	object	*		
updateVisu		method	*		

**Fig. 8.46** User-defined Attributes of the Container

Create the next steps in your tank element in the class library. The method updateVisu should be called when the value of fillingLevel changes. The best way to do this is to use an observer. Select Tools—Select observers. The attribute to observe is “fillingLevel”; if this changes, the method updateVisu is to be called. The setting in the observer dialog should look like in Fig. 8.47.

Observed Value	Method	Created here
fillingLevel	self.updateVisu	*

**Fig. 8.47** Observer Setting

The method updateVisu sets the icons according to the ratio between the filling level and the maximum filling level.

```
(Attribut: string; oldValue: any)
is
    symbolNo:integer;
do
    --set symbolNo acc. to % max. filling level
    symbolNo:=
        ceil((self.~.fillingLevel/self.~.maxFilling*10));
    if symbolNo /= self.~.currIconNo then
        self.~.currIconNo:=symbolNo; --show new icon
    end;
    --tank is empty --> stop
    if self.~.fillingLevel < 0 then
        messageBox(self.~.name + " is empty!",1,1);
        root.eventController.stop;
    end;
end;
```

You can now use the tank objects in the simulation. Create two entities for mapping the different products (Part1 and Part2). Create for the parts differently

colored icons. Ensure the following settings in the model: source: interval of one minute; MU-Selection: random, Table: CreationTable; insert into the CreationTable a distribution like in Fig. 8.48.

	object 1	real 2	integer 3	string 4
string	MU	Frequency	Number	Name
1	.MUs.Part1	80.00		
2	.MUs.Part2	20.00		

**Fig. 8.48** CreationTable

The line has a speed of 0.0167 m/sec. Create two sensors at the positions B1 and B2 on the line. Assign to both sensors the method Consumption as a control. Set the tank B2 as the reserveTank for B1 (and B4 for B3). You require a list of consumption values per part and tank as well as the replacement (delivery) times. Fill the table partData according to Fig. 8.49.

	string 0	real 1	real 2	time 3
string	Tank	Part2	Part1	Deliverytime
1	B1	50.00	75.00	2:00:00:00.0000
2	B3	30.00	50.00	2:00:00:00.0000

**Fig. 8.49** PartData

Note that the tank names are the row indices and the part names column indices. This simplifies the programming. The orderList should contain the tank (as an object reference), the date of the order and the respective delivery time. This requires formatting according to Fig. 8.50.

	object 1	time 2	time 3
string	Tank	Date of order	Delivery time
1			

**Fig. 8.50** Order List

To support the parameterization, the sensor list contains a mapping of the tanks to sensor numbers. In the simplest case, this appears as in Fig. 8.51 (row index = sensor number).

	string 1
string	Tank
1	B1
2	B3

**Fig. 8.51** SensorList

The init method deletes all orders and fills any tank up to its maxFilling:

```
is
do
  --fill all tanks
  B1.fillingLevel:=B1.maxFilling;
  B2.fillingLevel:=B2.maxFilling;
  B3.fillingLevel:=B3.maxFilling;
  B4.fillingLevel:=B4.maxFilling;
  --delete orders
  orderlist.delete;
end;
```

### Consumption

If a part triggers a sensor on the conveyor line, the consumption of material of this part is removed from the filling level of the tank at the sensor position. If the tank level falls below a liter, the order will be triggered automatically (by creating an entry in the order list). The conveyor line is stopped and the pumping over from the second tank is started. Since the pumping itself is not considered, it is enough to replenish the stock after a specified time to the maximum filling level. If the reserve tank is still empty because the delivery has not arrived yet, the simulation is terminated with an error message (the delivery time must be reduced or the tank size has to be adapted). This results in the following programming in the method Consumption:

```
(SensorID : integer; rear : boolean)
is
  tank:string;
  tankObj:object;
  consumption:real;
do
  -- read tank from sensorList
  tank:=sensorlist[1,SensorID];
  -- read consumption from partData
  consumption:=partData[@.name,tank];
  --subtract from tank filling level
  tankObj:=str_to_Obj(tank);
  tankObj.fillingLevel:=
    tankObj.fillingLevel-consumption/1000;
```

```

if tankObj.fillingLevel < 1 then
  --fail line for one hour
  productionLine.startFailure(time_pumping);
  --trigger order
  orderlist.writeRow(1,
  orderlist.yDim+1,
  tankObj.eventController.simTime,
  partData["Deliverytime",tank]);
  --start pumping
  wait(time_pumping);
  if tankObj.reserveTank.fillingLevel > 0 then
    tankObj.reserveTank.fillingLevel:=0;
    tankObj.fillingLevel:=tankObj.maxfilling;
  else
    MessageBox(tankObj.reserveTank.name+
      " is empty!",1,1);
    eventcontroller.stop;
  end;
end;
end;

```

## Delivery

Especially when the transport from the supplier to the company should not be shown in the simulation, it is sufficient if the ordered materials are available again after the delivery time (perhaps with a random scattering of the delivery time). The simulation tests enable determining whether the tank sizes are sufficient or the delivery time has to be shortened. For this realization, you may use a combination of generator and method. The generator calls the method at a regular interval. The method checks, based on the timing of the order and the delivery time, which delivery must arrive at the appropriate time (EventController.simTime) at the point of consumption and fills, in this example, the reserve tank. Delivered orders are removed from the order list. Set for the generator an interval of one minute. Assign the method Delivery as interval control. The method delivery has the following content:

```

is
  end_time;
  i:integer;
do
  --search in the orderlist
  --after deliverytime, refill reserve tank
  --and delete entry in the orderlist
  i:=1;
  if orderlist.yDim>0 then
    while(true) loop
      if (orderlist[2,i]+orderlist[3,i])<=
        eventcontroller.simtime then

```

```

--the order is due
orderlist[1,i].reserveTank.fillingLevel:=
orderlist[1,i].reserveTank.maxFilling;
--delete row
orderlist.cutRow(i);
--entries left ?
if orderlist.yDim = 0 then
    exitLoop;
end;
else
    i:=i+1;
    if orderlist[1,i] = void then
        exitLoop;
    end;
end;
end;
end;

```

The chart can be used to visualize the filling levels of the tanks (Fig. 8.52).

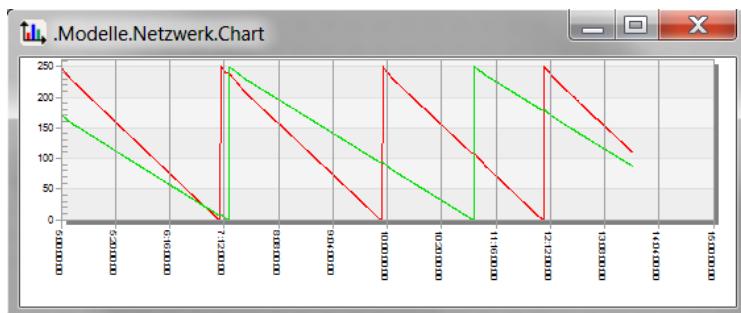


Fig. 8.52 Tank-filling Levels

#### 8.4.2.3 Goods Receipt Warehouse, Reorder Point Method

For the reorder point method, we need to expand a warehouse model by an order component. In this context, an inventory model is developed, which works without the "physical" storage of MUs and its control is based on a SQLite database. The delivery takes place in this model again without the concrete modeling of transport processes.

##### Virtual Warehouse—Modeling Approach (Database Supported)

If large warehousing systems need to be simulated and the warehouse technology is not the subject of the simulation, you can do this without the storage and retrieval of MUs in a store block. Especially when you need to transport many parts in containers and "hold" these parts in the simulation, this quickly leads to a significant deterioration of the simulation speed. Instead, you can destroy the MUs on entry into the warehouse and generate them again at retrieval. Storage location,

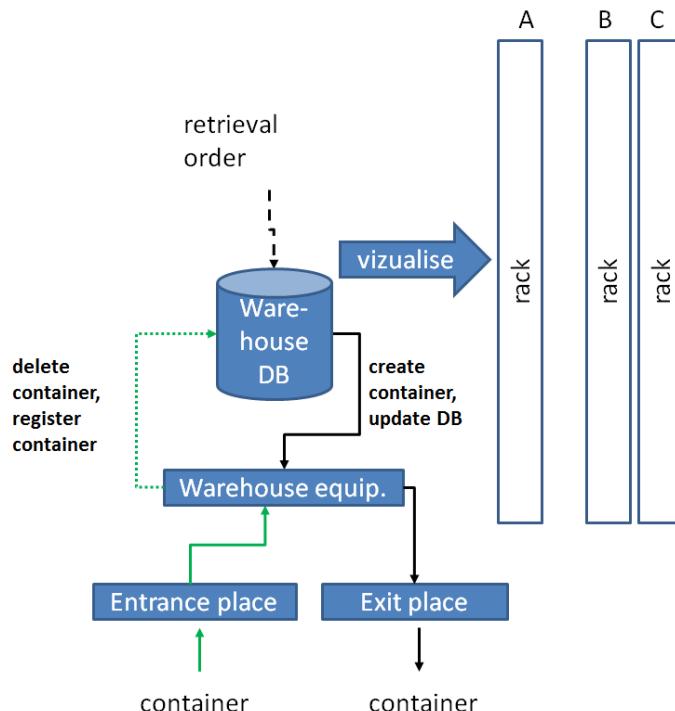
container and part type can be stored in a table (or database). For visualization and control, you can use simple table blocks. The use of databases here has the advantage that relatively complex data structures and logic can be realized.

### Warehouse Model

For the modeling of the warehouse, the following structure is recommended:

- Entry place (receives container, looking for free place in the warehouse, registers the container in the warehouse, destroys the container) with processing time
- Exit place (provides the container for the successor)
- Warehouse (visualizes the stock using a series of tables) consisting of a series of shelves
- Database (stores receipts with container and part type, number, storage location, entry time, etc.)
- Warehouse statistics (includes, among other things, part-specific statistics, such as incoming and outgoing quantities, number of stock movements)
- Warehouse control (realizes retrievals)

Schematically, this results in a model according Fig. 8.53 Schematic .



**Fig. 8.53** Schematic Model

The warehouse as a library element (network) requires elements like in Fig. 8.54 (create the shelves on the base of TableFiles).

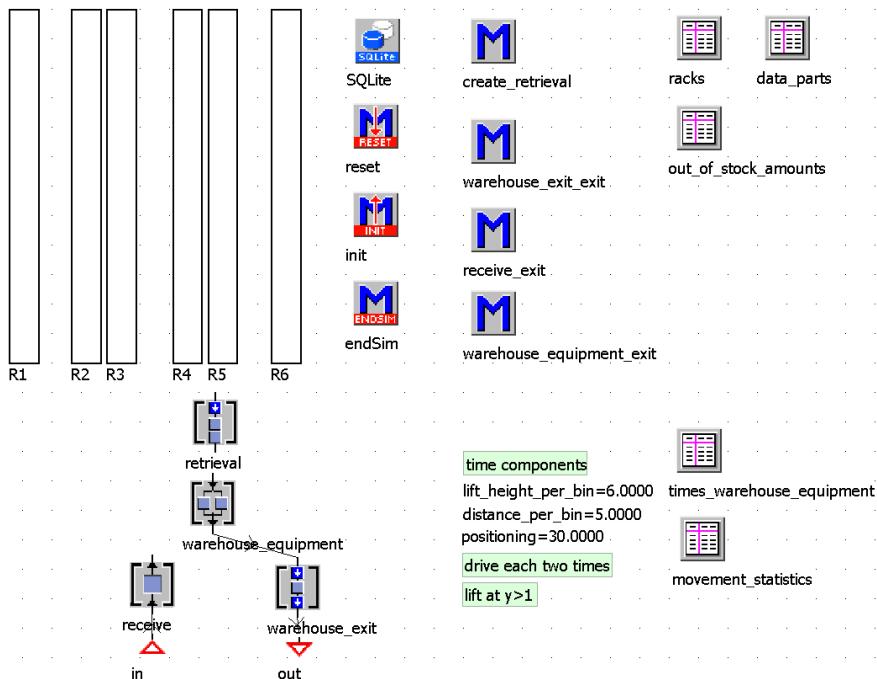


Fig. 8.54 Warehouse Sub-frame

### Modeling Approach for Warehouse Equipment

The storage equipment should not be modeled in detail (e.g. with tracks and transporter) in this model, but only through temporary occupation. Both storage and retrieval activities occupy storage technology and bind certain time resources. An option that is to be shown here is the use of a ParallelProc with place-dependent processing times. Before repositioning the container on the warehouse equipment, the time necessary for storage and retrieval is calculated. Depending on the destination of the container, it is stored (destroyed) or moved to the exit of the warehouse. The statistics of the ParallelProc then provides information about the necessary number of resources. Place the first four places in the ParallelProc warehouse\_equipment (x-dimension: 4, y-dimension: 1). Prepare settings like in Fig. 8.55 in the station warehouse\_equipment.

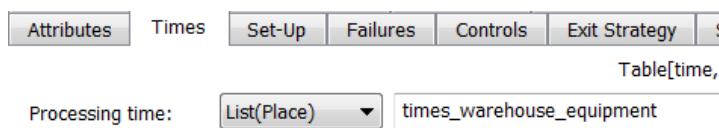


Fig. 8.55 Settings for Warehouse Equipment

The table is then formatted (one row, all columns of data type time). The column numbers correspond to the x-values of the places.

### Formatting of the TableFiles

The table movement statistics should contain for each part type, the quantity stored, the cumulative inflows and outflows and, for further analysis, the cumulative absences (for calculating the shortage costs). The details are given in Fig. 8.56.

	string 1	integer 2	integer 3	integer 4	time 5	integer 6
string Part	operation no.	cumul. in	cumul. out		cumul. waiting time (shortages)	number shortages
1						

Fig. 8.56 TableFile Formatting 1

The table out\_of\_stock\_amounts stores data on unsuccessful retrieval orders (Fig. 8.57).

	string 1	integer 2	object 3	datetime 4
string part	OP	destination		delayed since
1				

Fig. 8.57 TableFile formatting 2

The table data\_parts includes all data that are necessary for the disposition (Fig. 8.58).

string 0	integer 1	integer 2	integer 3	integer 4	real 5	real 6	integer 7	string 8	time 9	integer 10	object 11	object 12
string Part	OP	safety stock	order stock	order quantity	shortage costs	price	filling quantity	supplier	delivery time	warehouse area	part_MU	container_MU
1 Part1	0	2	15	30	1000.00	5.00	250	A	1:00:00:00.0000	3	.MUs.Part1	.MUs.Container
2 Part2	0	2	15	50	500.00	10.00	50	B	1:00:00:00.0000	1	.MUs.Part2	.MUs.Container
3 Part3	0	2	15	30	25.00	1000.00	100	C	1:00:00:00.0000	2	.MUs.Part3	.MUs.Container

Fig. 8.58 TableFile Formatting 3

Insert in the rack objects (TableFiles) two user-defined attributes for the dimensioning of the racks:

- xDimension (integer, value = 30)
- yDimension (integer, value = 6)

### Initializing

The init method deletes all values of the tables (except the table data\_parts), determines all racks located in the network and initializes the warehouse database, by entering all warehouse locations (defined in the rack tables) into the database. There is a peculiarity in the use of in-memory databases. The database exists only in the main memory. Before the database can be used, all of the database tables must be generated. After closing the database connection, the data are no longer available (that is why you need to write the statistics in a Plant Simulation table).

For inventory management and the later production of parts and containers, we need the following data (structure of the database table stock):

- ID (INTEGER, primary key)
- place\_rack (TEXT)
- place\_X (INTEGER)
- place\_Y (INTEGER)
- place\_category (INTEGER)
- part (TEXT)
- op (TEXT)
- quantity (INTEGER)
- part\_mu (TEXT)
- container\_mu (TEXT)
- entrance\_time (INTEGER)

The need to generate the stock table requires the following SQL command:

```
CREATE TABLE stock (ID INTEGER AUTO_INCREMENT PRIMARY KEY, place_rack TEXT, place_X INTEGER, place_Y INTEGER, place_category INTEGER, part TEXT, op TEXT, quantity INTEGER, part_mu TEXT, container_mu TEXT, entrance_time INTEGER)
```

All warehouse places are entered in the database table, in the first step with the rack, X and Y coordinates and place category.

As IDs, sequential numbers are assigned. In the field for the part, we use '-' as a marker that this place is empty. Leave all other fields empty. To enter a place, you will need the following SQL statement:

```
INSERT INTO stock (ID, place_rack, place_X, place_Y, part) VALUES (1, 'rack', x, y, '-')
```

This yields the following init method:

```
is
  i:integer;
  k:integer;
  m:integer;
  n:integer;  found:boolean;
  container:string;
  part:string;
  sql:string;
  place_category:integer;
do
  out_of_stock_amounts.delete;
  movement_statistics.delete;
  racks.delete;
  --determine all racks
  for i:=1 to self.~.numNodes loop
    if self.~.node(i).class.name="VRack" then
```

```

--insert into table row
racks.writeRow(1,racks.yDim+1,
self.~.node(i));
end;
next;
--create connection to the database
sqlite.open;
-- create table stock
sql:="CREATE TABLE stock (ID INTEGER"+
"AUTO_INCREMENT PRIMARY KEY, place_rack TEXT, "+
"place_X INTEGER,place_Y INTEGER, "+
" place_category INTEGER, part TEXT, op TEXT, "+
" quantity INTEGER, part_mu TEXT, "+
" container_mu TEXT, entrance_time INTEGER)";
sqlite.exec(sql);
--write all warehouse places into the table stock
n:=1;
for i:=1 to racks.yDim loop
  for k:=1 to racks[1,i].xDimension loop
    for m:=1 to racks[1,i].yDimension loop
      --read place categories k=x und m=y
      if k<5 and m<4 then
        place_category:=1;
      elseif k<8 and m<6 then
        place_category:=2;
      else
        place_category:=3;
      end;
      sql:="INSERT INTO stock (ID,place_rack, place_X, "+
place_Y,place_category,part) VALUES ("+
to_str(n)+", '"+racks[1,i].name+"', "+to_str(k)+", "+
to_str(m)+", "+to_str(place_category)+", '-' )";
      n:=n+1;
      sqlite.exec(sql);
    next;
  next;
next;
end;

```

Note: If you repeatedly call the init method, you will receive an error message "table stock already exists." The database is still open and the table still exists.

You must close the database after each call of init (e.g. by calling the reset method). The method reset:

```

is
do
  sqlite.close;
end;

```

### Generate Initial Stock in the Warehouse

At the beginning of a simulation run, the stock must be filled with the safety stock and the order quantity for each part. The easiest way is to "fill" the first places in the stock table with the parts in the init method. The storing is carried out by changing a record. You need an UPDATE statement—e.g.:

```

UPDATE stock SET part='part1', OP='0', quantity=250,
part_MU='MUs.part1', container_MU='MUs.container
WHERE ID=1

```

The opening stock must be created dynamically using the data of the table data\_parts. Extend the init method as follows (at the end of the method):

```

--initialize warehouse with parts from
--table parts_data
container:=".MUs.Container";
m:=1;
for i:=1 to data_parts.yDim loop
  for k:=1 to (data_parts[2,i]+data_parts[4,i]) loop
    part:=".MUs."+data_parts[0,i];
    --create SQL statement
    sql:="UPDATE stock SET part='"+data_parts[0,i]+
      "', OP='0', quantity="+to_str(data_parts[7,i])+"
      ", part_MU='MUs."+data_parts[0,i]+
      "', container_MU='"+container+
      "' WHERE ID="+to_str(m);
    sqlite.exec(sql);
    m:=m+1;
  next;
next;

```

### Test Container

The test container is a small section from a production line. At each station, parts are consumed. When a container is empty, a new container is requested from the warehouse (the empty container is destroyed). The place of the container has to take over a part of the information flow. Duplicate a store block, rename it as P and set custom attributes according to Fig. 8.59.

Name	Value	Type	C.	L.	3l
getPart		method	*		
OP	0	integer	*		
part		string	*		
reserve	(?)	object	*		
warehouse	(?)	object	*		

Fig. 8.59 User-defined Attributes of Container Place

## Two-container System

The supply of the process with parts should work as a two-container system. This means that the process takes parts from a defined container position. When the container is empty, the replenishment is requested. During the delivery time, the consumption takes place from a second container. The removal of the part will be effected with the method `getPart`. When the container is empty, it is destroyed and a new container is requested from the warehouse. The container from the reserve place is relocated to the takeoff point. In order to program it at the takeoff point, you need references to the reserve place, the warehouse and information about the part that is consumed in the operation. The order of the material in the warehouse is carried out by calling the method:

```
create_retrieval (part:string;op:integer;
                  destination:object).
```

The method `getPart` contains the subsequent programming:

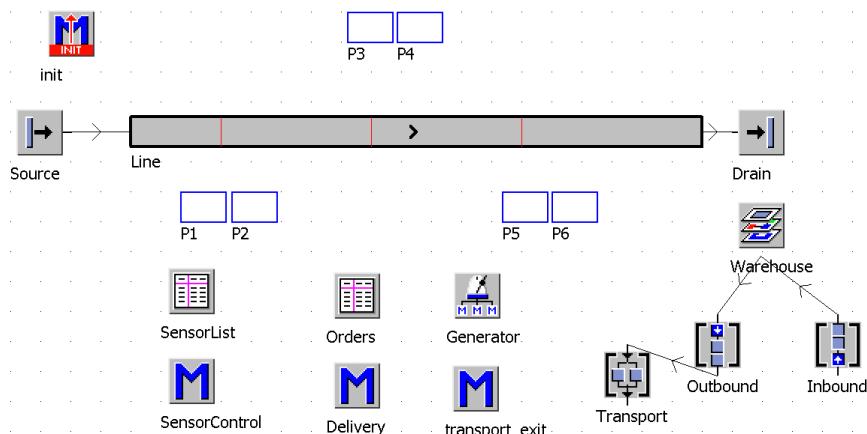
```
is
do
  --delete one part, abort if the is no part
  --or container
  if self.~.empty then
    messageBox("No container at "+self.~.name+
               " available!",1,1);
    eventController.stop;
  else
    --container empty ?
    if self.~.cont.empty then
      messageBox("Container at "+self.~.name+
                 " is empty!",1,1);
      eventController.stop;
    else
      -- delete one part
      self.~.cont.cont.deleteObject;
      -- container empty??
      if self.~.cont.empty then
        --request retrieval
        self.~.warehouse.create_retrieval(
          self.~.part,self.~.op,self.~.reserve);
        -- delete container
        self.~.cont.delete;
        -- if container at reserve, move
        -- otherwise error message
        if self.~.reserve.occupied then
          self.~.reserve.cont.move(self.~);
```

```

        else
            messageBox("No container available at "
                +self.~.reserve.name+"!",1,1);
            eventcontroller.stop;
        end;
    end;
end;
end;
end;

```

Create a frame like in Fig. 8.60. Insert into the frame the sub-frame warehouse. Create container places P1 to P6.



**Fig. 8.60** Example Frame

The source generates an assembly every minute (object container). The line has a speed of 0.1 meters per second. Include at three, eight and 13 meters sensors on the line, and assign to each SensorControl as the method. Prepare the following settings in the container places:

Attribute	P1	P2	P3	P4	P5	P6
Warehouse						
OP	0	0	0	0	0	0
Reserve	P2		P4		P6	
Part	Part1	Part1	Part2	Part2	Part3	Part3

The SensorList contains an assignment of container places to sensors (Fig. 8.61).

object	
string	Place
1	P1
2	P3
3	P5

**Fig. 8.61** SensorList

Format the table Orders according to Fig. 8.62.

	string	integer	string	integer	integer	time	time
string	Part	OP	supplier	quantity_containers	fill amount	order date	delivery time
1							

**Fig. 8.62** Table Orders

Insert in the container in the class library a user-defined attribute access\_time of the data type time.

When an MU triggers the sensor control, the internal method getPart in the container place should be called. For this, you must first identify which container place is located at this position (using theSensorList and the SensorID passed through Plant Simulation). The method SensorControl has the following content.

```
(SensorID : integer; Rear : boolean)
is
  place:object;
do
  place:=SensorList[1, SensorID];
  --consume part
  place.getPart;
end;
```

When a container is empty, the control getPart calls the method create\_retrieval of the warehouse. The method needs the following initial form:

```
(part:string; op:integer; destination:object)
is
do
end;
```

#### 8.4.2.3.1 Warehouse Retrievals

The containers and parts are only stored as data in the database. Therefore, both of the containers and the parts must be generated in the necessary amount prior to the actual retrieval. The storage technology is modeled using a parallel station.

Depending on the position of the parts in the warehouse, the processing time of one place is set. In this way, the need for material handling capacity can be checked. The basic sequence of retrieval of parts from a virtual warehouse is shown in Fig. 8.63.

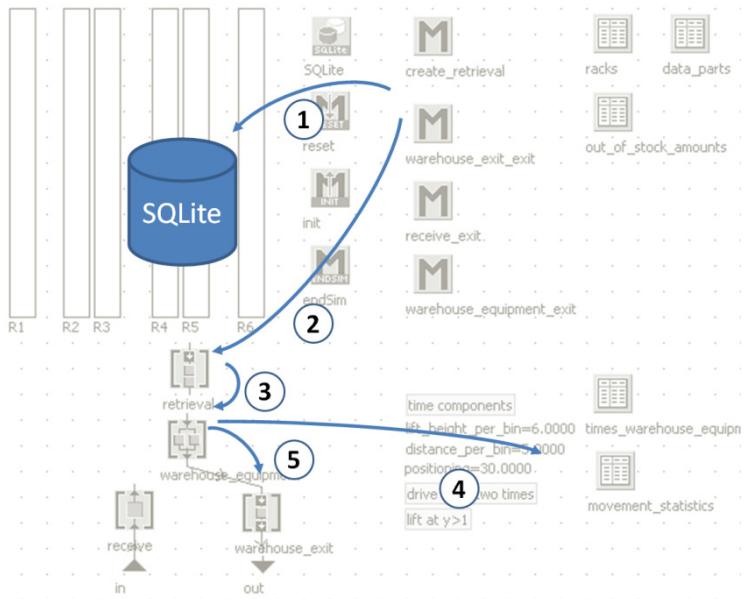


Fig. 8.63 Retrieval Process

- In the database, the requested part is searched for in the required OP.
- A container and the requested parts are generated in the buffer retrieval. The entry in the database is updated (the place is marked as empty). If the stock has dropped below the reorder level and no entry for the part is entered in the table Orders, an order is created by through an entry in the table Orders.
- The exit control transfers the containers to warehouse\_equipment if capacity is available. The individual pick-time is considered as the processing time of one place of the Parallel Station.
- The retrieval is registered in the Statistics table.
- After the process time of the warehouse equipment, the container is relocated to the warehouse\_exit.

### Find and Provide Parts, Generate Orders

The programming is mainly located in the method `create_retrieval`. The rough flow into the programming is as follows:

- You need to search for a part with the correct OP. This is done with an SQL query—e.g. `SELECT * FROM stock WHERE part='part2' and OP='0' ORDER BY entrance_time DESC LIMIT 1`. The

descending sort (ORDER BY ... DESC) by the entrance time ensures a FIFO principle at the retrieval. With the LIMIT = 1 option the query returns only one record.

2. The place will be marked as empty ('-'). For this purpose, an UPDATE statement is used as in "UPDATE stock set part = '-' WHERE ID = 1". Thus, the same part is not "found" twice.
3. The remaining stock of this part is checked. You can use for this the query:  
SELECT COUNT(id) FROM stock WHERE part='part2' and OP = '0'.
4. If the stock is below the reorder point and no order is on the way, a order is triggered.
5. If a place is available, a container is created and filled with parts corresponding to the data from the database.
6. According to the "virtual" storage place, an accessing time is calculated and assigned to the container.

```
(part:string;op:integer; destination:object)
is
  i,x,y,id,number,stock:integer;
  t:time;
  sql:string;
  part_obj,container_obj:object;
  container,part_:object;
  found:boolean;
do
  --search for the part in the database (1)
  sql:="SELECT * FROM stock WHERE part='"+part+
    "' and OP='"+to_str(op)+"
    "' ORDER BY entrance_time ASC LIMIT 1";
  sqlite.prepare(sql);
  if sqlite.step then
    id:=sqlite.getColumnInteger(0);
    x:=sqlite.getColumnInteger(2);
    y:=sqlite.getColumnInteger(3);
    number:=sqlite.getColumnInteger(7);
    part_obj:=str_to_obj(sqlite.getColumnString(8));
    container_obj:=
      str_to_obj(sqlite.getColumnString(9));
    sqlite.finalize;
    --mark place as empty (2)
    sql:="UPDATE stock set part='-' WHERE ID="+
      to_str(id);
    sqlite.exec(sql);
    --check stock, if below reorder point (3)
    --then trigger order
    --if not exists an oder (part+OP)
    sql:="SELECT COUNT(id) FROM stock WHERE part=''"
```

```
+part+"' and OP='"+to_str(op)+"'";
sqlite.prepare(sql);
sqlite.step;
stock:=sqlite.getColumnInteger(0);
if stock <= data_parts[3,part] then
    --create order
    found:=false;
    for i:=1 to root.orders.yDim loop
        if root.orders[1,i]=part
            and root.orders[2,i]=op then
            found:=true;
            exitLoop;
        end;
    next;
    if not found then
        -- insert one row into the table orders (4)
        root.orders.writeRow(1,root.orders.yDim+1,
            part,op,data_parts[8,part],
            data_parts[4,part],data_parts[7,part],
            eventcontroller.simTime,
            data_parts[9,part]);
    end;
end;
--retrieval --> create container and parts (5)
waituntil retrieval.full=false prio 1;
container:=container_obj.create(retrieval);
container.xDim:=number;
container.destination:=destination;
for i:=1 to number loop
    part_:=part_obj.create(container);
    part_.op:=op;
next;
--calculate access time (6)
t:=positioning+(y-1)*lift_height_per_bin+
    x*distance_per_bin;
container.access_time:=t;
else
    sqlite.finalize;
    --out of stock
    --enter out of stock, if not already exists
    out_of_stock_amounts.writeRow(1,
        out_of_stock_amounts.yDim+1,part,op,
        destination,eventController.absSimTime);
    messageBox(part+ " out of stock!",1,1);
    eventcontroller.stop;
end;
end;
```

### Setting up the Warehouse Equipment

The capacity requirements for warehouse technique can be represented using a ParallelProc, whose stations have different processing times. The removal and storage functions respectively compute times for storage and retrieval depending on the storage location. Before moving the container, this time must be entered in the table “times\_warehouse\_equipment” corresponding to the free place of the ParallelProc “warehouse\_equipment” (exit control in front of retrieval—method “warehouse\_exit\_exit”). Set the capacity of the “warehouse\_equipment” to four (x = 4, y = 1). In the method “warehouse\_exit\_exit,” first a free place is sought, then the time for this place is set. Next, the container is moved to “warehouse\_equipment.” This yields the following programming:

```

is
  i:integer;
  place:integer;
do
  waituntil warehouse_equipment.full=false prio 1;
  --look for a free place
  for i:=1 to warehouse_equipment.xDim loop
    if warehouse_equipment.pe(i,1).cont=void then
      place:=i;
      exitLoop;
    end;
  next;
  -- set procTime for this place
  times_warehouse_equipment[place,1]:=@.access_time;
  --move
  @.move(warehouse_equipment.pe(place,1));
end;

```

The warehouse technology destroys containers to be stored and moves the remaining container to warehouse\_exit (exit control front, method warehouse\_equipment\_exit). Storage and retrieval are registered.

```

is
  i:integer;
  found:boolean;
do
  if @.destination=void then
    --register storage
    for i:=1 to movement_statistics.yDim loop
      if movement_statistics[1,i]=@.inhalt.name and
        movement_statistics[2,i]= @.cont.op then
        movement_statistics[3,i]:=_
          movement_statistics[3,i]+1;
        exitLoop;
      end;
    next;

```

```

@.delete;
else
  --register retrieval
  --search for part and op
  for i:=1 to movement_statistics.yDim loop
    if movement_statistics[1,i]=@.cont.name and
      movement_statistics[2,i]= @.cont.op then
      movement_statistics[4,i]:= 
        movement_statistics[4,i]+1;
      found:=true;
      exitLoop;
    end;
  next;
  --if not found - new row
  if found=false then
    movement_statistics.writeRow(1,
      movement_statistics.yDim+1,
      @.cont.name,@.cont.op,0,1);
  end;
  @.insert(self.~.succ);
end;
end;

```

### Transport to the Point of Consumption

The transport is simulated similarly to the warehouse equipment as a parallel station (with an average transport time as processing time). In the exit control of a transport-parallel station, the containers are relocated to their final destination. The exit control (method `transport_exit`) is as follows:

```

is
do
  @.move(@.destination);
end;

```

#### 8.4.2.3.2 Deliveries

The warehouse management system creates orders if the stock falls below the reorder point (table Orders; see Fig. 8.64).

	string 1	integer 2	string 3	integer 4	integer 5	time 6	time 7
string	Part	OP	supplier	quantity_containers	fill amount	order date	delivery time
1	Part3	0	C	30	100	11:19:31:40.0000	1:00:00:00.0000
2	Part2	0	B	50	50	12:20:23:20.0000	1:00:00:00.0000

**Fig. 8.64** Orders

In this case, the transport from the supplier must not be considered. It is important only that, after the delivery time, the parts arrive at the goods receipt.

The delivery time could be provided with a random scattering, to better represent the behavior of the real system. In order to realize this, one method involves looking in a regular time interval (called by a generator) in the order list for "due" deliveries. Containers to be delivered are generated directly in the inbound. The necessary data are located in the order list and in the table data\_parts in the warehouse sub-frame. The method delivery (called in the interval of e.g. one minute from the generator) would look like this:

```

is
  end_time;
  i,m,n:integer;
  part_MU:object;
  container:object;
  part:string;
do
  --look in orders
  --after delivery time, create container in outbound
  -- fill with parts
  --delete entry in orders
  i:=1;
  if orders.yDim>0 then
    while(true) loop
      if (orders[6,i]+orders[7,i])
        <=eventController.simTime then
        --order is due
        part:=orders[1,i];
        --create container
        for m:=1 to orders[4,i] loop
          container:=
            warehouse.data_parts["Container_MU",
              part].create(inbound);
          --set capacity
          container.xDim:=orders[5,i];
          --create parts
          for n:=1 to orders[5,i] loop
            part_MU:=warehouse.data_parts["Part_MU",
              part].create(container);
            part_MU.op:=orders[2,i];
          next;
        next;
        --remove row
        orders.cutRow(i);
        --remaining entries?
        if orders.yDim = 0 then
          exitLoop;
        end;
      else

```

```

i:=i+1;
  if orders[1,i] = void then
    exitLoop;
  end;
end;
end;
end;

```

#### 8.4.2.3.3 Inbound, Out-of-stock Part

If the part is registered in the table `out_of_stock_amounts`, it will not be transported into the warehouse. It is directly moved to the warehouse exit and is provided immediately for production (Fig. 8.65).

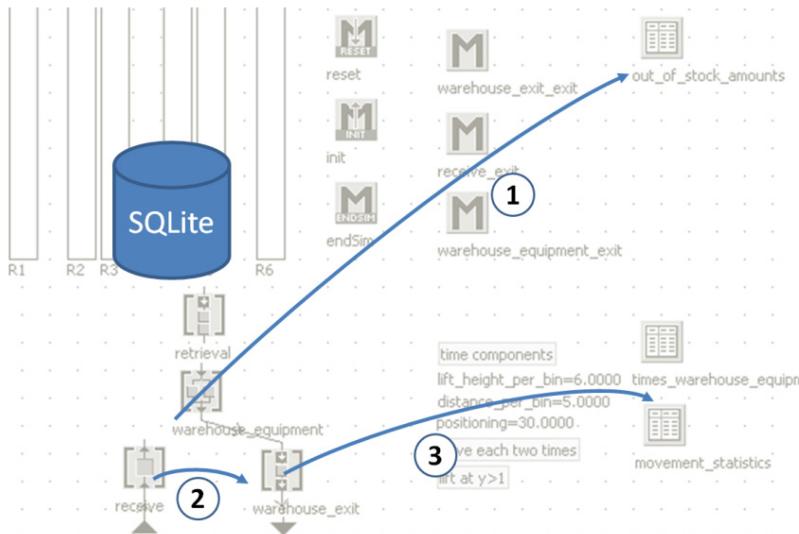


Fig. 8.65 Handling of Out-of-stock Part

1. For the container, it is checked whether an entry in the table `out_of_stock_amounts` exists (part, OP).
2. The container is moved directly to `warehouse_exit`.
3. The out-of-stock duration is stored in the table `movement_statistics`.

This results in the following programming in the method `receive_exit`:

```

is
  rack:string;
  x,y,id,i,place:integer;
  found:boolean;
  t:time;

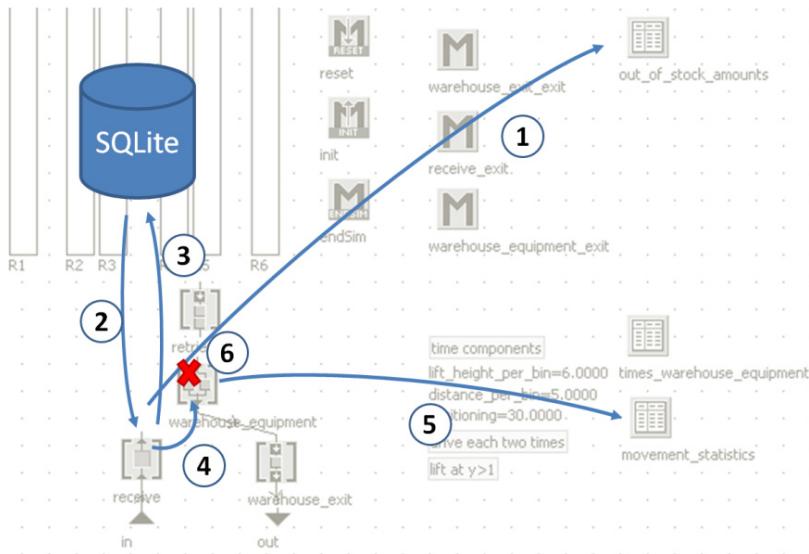
```

```

sql:string;
zone:integer;
do
  --out of stock-handling
  -- search part and op in out of stock table
  for i:=1 to out_of_stock_amounts.yDim loop
    if out_of_stock_amounts[1,i]=@.cont.name and
       out_of_stock_amounts[2,i]=@.cont.op then
      --out of stock
      waituntil warehouse_exit.empty prio 1;
      --update statistics
      --look for the part in the first column
      movement_statistics.setCursor(1,1);
      found:=movement_statistics.find({1,1}..{1,*},
                                       @.cont.name);
      --if not found --> enter
      if not found then
        movement_statistics[1,
                           movement_statistics.yDim+1]:=part;
        movement_statistics.setCursor(1,
                                       movement_statistics.yDim);
      end;
      --out of stock time
      t:=eventController.absSimTime-
          out_of_stock_amounts[4,i];
      movement_statistics[5,
                         movement_statistics.zeigerY]:=-
                         movement_statistics[5,
                         movement_statistics.zeigerY]+zeit;
      --number out of stock events
      movement_statistics[6,
                         movement_statistics.zeigerY]:=-
                         movement_statistics[6,
                         movement_statistics.zeigerY]+1;
      --adress container
      @.destination:=out_of_stock_amounts[3,i];
      --delete row in out of stock amounts
      out_of_stock_amounts.cutRow(i);
      --move container
      @.move(warehouse_exit);
      --end
      return;
    end;
  next;
  --look for a free warehouse place
end;

```

If there is no out-of-stock entry, then the container is registered in the warehouse before being transferred to the warehouse technology (which destroys it). The inbound process takes place in several steps analogous to Fig. 8.66.



**Fig. 8.66** Inbound Process

1. It is checked whether an out-of-stock event is registered for the part (possibly out of stock handling).
2. An empty place in the warehouse database is searched.
3. The container is registered as a record in the database.
4. An inbound time will be set and the container relocated to the warehouse equipment.
5. The storage will be registered in the table movement\_statistics.
6. The container will be destroyed (warehouse equipment).

The programming is located in the method receive\_exit (below the out-of-stock handling).

```

is
  rack:string;
  x,y,id,i,place:integer;
  found:boolean;
  t:time;
  sql:string;
  zone:integer;
do
  --out of stock-handling (1)

```

```

-----
--look for a free warehouse place (2)
--empty places are marked with '-'
sql:="SELECT ID,place_X,place_Y from stock WHERE
art='-' LIMIT 1";
sqlite.prepare(sql);
if sqlite.step then
  id:=sqlite.getColumnInteger(0);
  x:=sqlite.getColumnInteger(1);
  y:=sqlite.getColumnInteger(2);
  sqlite.finalize;
--calculate inbound time
t:=positioning+(y-1)*lift_height_per_bin+
  x* distance_per_bin;
--register part, container (3)
sql:="UPDATE stock set part='"+@.cont.name+
  "' , OP='"+to_str(@.cont.op)+"', quantity="+
  to_str(@.numMu)+",part_MU='"+
  to_str(@.cont.class)+"', "+
  "container_MU='"+to_str(@.class)+"
  "' , entrance_time="+
  to_str(time_to_num(eventController.simTime))+"
  " WHERE ID='"+to_str(id);
sqlite.exec(sql);
@.destination:=void;
--if place is available, set time and move (4)
waituntil warehouse_equipment.full=false prio 1;
--look for a free place
for i:=1 to warehouse_equipment.xDim loop
  if warehouse_equipment.pe(i,1).cont=void then
    place:=i;
    exitLoop;
  end;
next;
--set ProcTime for this place
times_warehouse_equipment[place,1]:=t;
--move
@.move(warehouse_equipment.pe(i,1));
else
  --warehouse is full
  messageBox("warehouse "+self.~.name+
    " is full!",1,1);
  eventController.stop;
  sqlite.finalize;
end;
end;

```

#### 8.4.2.3.4 Visualization of the Database Stock

A disadvantage of working with in-memory databases is that the values of the tables in the simulation are not visible. For control, it is useful to create visualizations. The easiest way is to write all entries of a database table into a Plant Simulation table. For example, you could use a button to start a method (see elements in Fig. 8.67).



**Fig. 8.67** Elements showStock

The method showStock reads all data from the SQLite table stock and writes them cell by cell in the table SQLite\_stock. So that old values do not remain in the table, all the old values in the table are deleted first.

```

is
  sql:string;
  i:integer;
  cols:integer;
do
  sqlite_stock.delete;
  --read all data
  sql:="SELECT * FROM stock";
  sqlite.prepare(sql);
  while(sqlite.step) loop
    --write cell by cell
    for i:=0 to sqlite.getColumnCount-1 loop
      if i=0 then
        sqlite_stock[1,sqlite_stock.yDim+1]:=sqlite.getColumnString(i);
      else
        sqlite_stock[i+1,sqlite_stock.yDim]:=sqlite.getColumnString(i);
      end;
    next;
  end;
end;

```

Another possibility is visualization with the help of tables. Using tables, you can easily show a schema of the storage racks. With maxXDim and maxYDim, you can limit the displayed number of rows and columns. Now, you only have to read for each rack the parts stored in the database and enter it into the table at the respective x, y position. For a better overview, you can change the background color of the "occupied" table cells. To make the call to the user comfortable, you

can run the appropriate method when opening the "rack" items. To do this, insert in the class of the rack (VRack in the class library) a user-defined method attribute (onOpen). This method is assigned to the table as control, which is called on opening (see Fig. 8.68, Tools—Select controls).

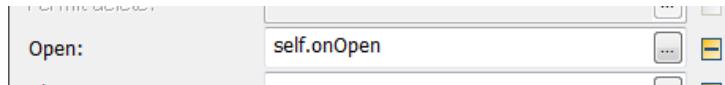


Fig. 8.68 Open Control

The method onOpen adjusts as a first step the visible dimension of the table. All dates for this rack are then read from the database. If the warehouse place is empty ('-'), then the background color of the cell is set to white; otherwise, it is set to light blue.

```

is
  sql:string;
  x:integer;
  y:integer;
do
  --set maxXDim and maxYDim after the dimension
  --of the rack
  self.~.openDialog;
  self.~.maxXDim:=self.~.xDimension;
  self.~.maxYDim:=self.~.yDimension;
  --read all places of the rack
  sql:="SELECT place_X,place_Y,part from stock "+
    "WHERE place_rack='"+self.~.name+"' ";
  --write all parts into the table
  --invert Y coordinates (YDim-Y+1)
  sqlite.prepare(sql);
  while(sqlite.step) loop
    x:=sqlite.getColumnInteger(0);
    y:=self.~.yDimension+1sqlite.getColumnInteger(1);
    (self.~)[x,y]:=sqlite.getColumnString(2);
    if (self.~)[x,y] = "-" then
      self.~.setBackgroundColorCells(
        {x,y},makeRGBValue(255,255,255));
    else
      self.~.setBackgroundColorCells(
        {x,y},makeRGBValue(95,243,251));
    end;
  end;
  sqlite.finalize;
end;

```

The Y positions are changed so that the position y=1 corresponds to the position at the bottom of the rack. Thus, the table shows the stored occupation of the rack (Fig. 8.69).

	string 1	string 2	string 3	string 4	string 5	string 6	string 7	string 8	string 9	s 1
1	Part1	-	-							
2	Part3	Part1	Part1	-						
3	Part3	Part3	Part3	Part3	Part3	-	Part3	Part1	Part1	-
4	-	-	-	-	Part3	Part3	Part3	Part1	Part1	-
5	-	-	-	-	Part3	Part3	Part3	Part1	Part1	-
6	-	-	-	-	Part3	Part3	Part3	Part1	Part1	-

**Fig. 8.69** Rack Visualization

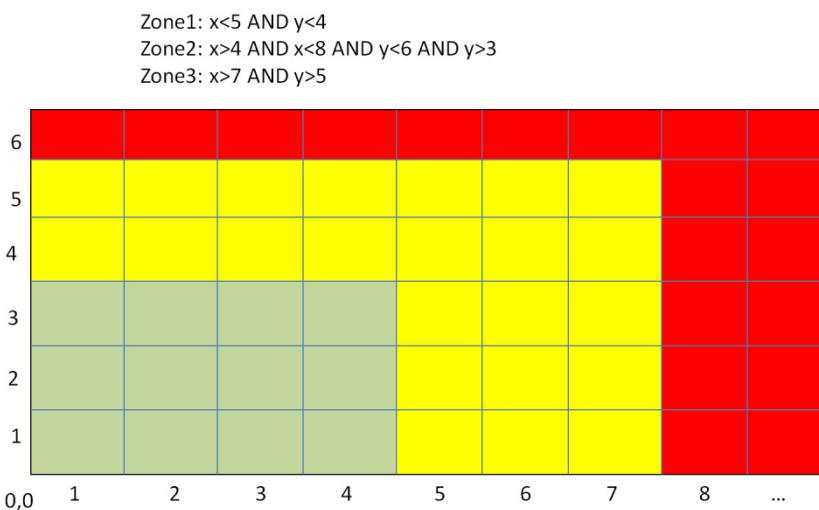
#### 8.4.2.3.5 Storage and Retrieval Orders

You can adjust the basic behavior of the warehouse relatively easily with the help of SQL. Two cases should be considered:

- Establishment of warehouse zones to optimize access times (storage)
- First-in first-out (retrieval)

#### Warehouse Zones

To optimize access times, the racks can be split in zones (see Fig. 8.70).



**Fig. 8.70** Warehouse Zones

For the simulation of storage zones, some steps must be taken:

1. Assignment of each place to a warehouse zone (init method)
2. Assignment of parts to warehouse zones (Table data\_parts)
3. Account of warehouse zone when storage (method receive\_exit)

(1) When you create the warehouse places in the database (init method), each place is assigned to a zone (field place\_category). Without parameterizing the logic of the zone mapping, it would look like this (excerpt from the init method).

```
--write all warehouse places into the table stock
n:=1;
for i:=1 to racks.yDim loop
  for k:=1 to racks[1,i].xDimension loop
    for m:=1 to racks[1,i].yDimension loop
      --read place categories k=x und m=y
      if k<5 and m<4 then
        place_category:=1;
      elseif k<8 and m<6 then
        place_category:=2;
      else
        place_category:=3;
      end;
      sql:="INSERT INTO stock (ID,place_rack, place_X, "+place_Y,place_category,part) VALUES ("+
      to_str(n)+", '"+racks[1,i].name+"','"+to_str(k)+"','"+to_str(m)+"','"+to_str(place_category)+"', '-' )";
      n:=n+1;
      sqlite.exec(sql);
    next;
  next;
next;
```

(2) In the table data\_parts, the following assignments are stored (column 10, warehouse area, integer): part1—zone 3, part2—zone1 and part3—zone 3.

(3) The warehouse zone must be integrated into the query to search an empty place. Here, a place is searched, with the value in the field place\_category greater than or equal to the value from the part in the data\_parts column 10. The search result is sorted so that the record with the smallest value in warehouse zone is returned. Excerpt from method receive\_exit:

```
sql:="SELECT ID,place_X,place_Y from stock WHERE
part='-' AND place_category>='"+to_str(zone)+" ORDER BY
place_category ASC LIMIT 1";
```

### **Retrieval Order First In, First Out (FIFO)**

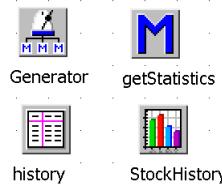
The retrieval order is controlled by the SQL statement in the method create\_retrieval. The query must be extended for this so that the container with the longest dwell time in stock (thus, the lowest entrance time → sort by entrance\_time or sub-query) is searched. The query would look like this (version sorting):

```
sql:="SELECT * FROM stock WHERE part='"+part+
" and OP='"+to_str(op)+"
 ORDER BY entrance_time ASC LIMIT 1";
```

Just as easily, you can create last in, first out (LIFO) queries (sort descending). Database-supported warehouse controls are very flexible and easy to customize for more complex tasks.

#### 8.4.2.3.6 Warehouse Statistics (Database)

Stock statistics (mainly stock history) must periodically be extracted from the database (generator; distance: 10 minutes; distance control: method `get_statistics`) and summarized in a table (History) in Plant Simulation. Complete the warehouse class with objects for the statistics according to Fig. 8.71.



**Fig. 8.71** Warehouse Statistics

The stock of containers with part1 you result from the database with the query:

```
SELECT COUNT(ID) FROM stock WHERE part='part1'
```

The query returns only one value. The entire method `get_statistics` might look like this:

```

is
  sql:string;
  num_part1:integer;
  num_part2:integer;
  num_part3:integer;
do
  --read number of containers for part1,
  --part2 and part3
  sql:-
  "SELECT COUNT(ID) FROM stock WHERE part='Part1'";
  sqlite.prepare(sql);
  if sqlite.step then
    num_part1:=sqlite.getColumnInteger(0);
  end;
  sqlite.finalize;
  sql:-
  "SELECT COUNT(ID) FROM stock WHERE part='Part2'";
  sqlite.prepare(sql);
  if sqlite.step then
    num_part2:=sqlite.getColumnInteger(0);
  end;

```

```

sqlite.finalize;
sql:-
"SELECT COUNT(ID) FROM stock WHERE part='Part3'";
sqlite.prepare(sql);
if sqlite.step then
  num_part3:=sqlite.getColumnInteger(0);
end;
sqlite.finalize;
--write row in table history
history.writeRow(1,
  history.yDim+1,num_part1,num_part2,num_part3);
end;

```

The recorded values can then be easily represented in Plant Simulation with the help of a chart object (Fig. 8.72).



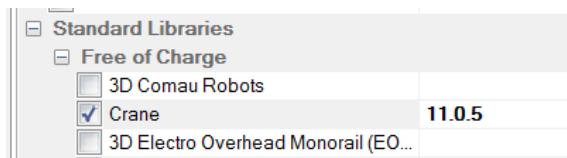
Fig. 8.72 Stock chart

### 8.4.3 The StorageCrane Object

Plant Simulation provides through the StorageCrane an object to simulate area storages, which are loaded and unloaded by a portal crane in a simple way. The crane itself has a storage and retrieval place where it can pick and place MUs, and a certain amount of storage places are arranged in the x, y, z direction (where the z-axis denotes the stack height). Use standard strategies provided by Plant Simulation for storage and retrieval (after a certain time) or formulate your own strategies.

#### 8.4.3.1 Store and Remove Automatically with the StorageCrane

To use the StorageCrane, you must first import the PortalCranes library into your model. Select File—Manage Class Library. You will find the Cranes in Libraries—Standard Libraries (Fig. 8.73).



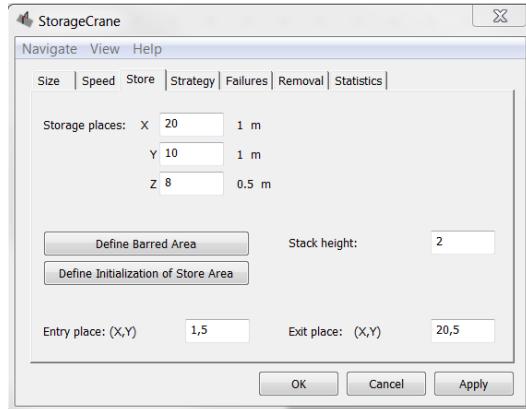
**Fig. 8.73** Crane Library

Create with the StorageCrane a simple model like in Fig. 8.74.



**Fig. 8.74** Example Frame

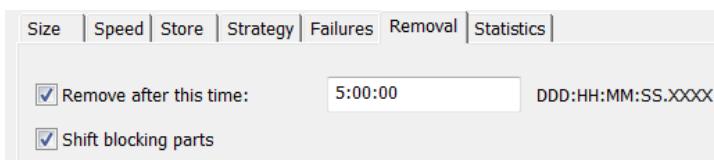
Place all objects in their default settings. Set the size of the crane in its dialogue box (crane length, portal width and portal height). In the tab Store, set the number of store places (x, y, z). With stack height, determine how many MUs the crane stacks one above the other (Fig. 8.75). With the barred area, define an area that must not occupy the crane.



**Fig. 8.75** StoreCrane Dialog

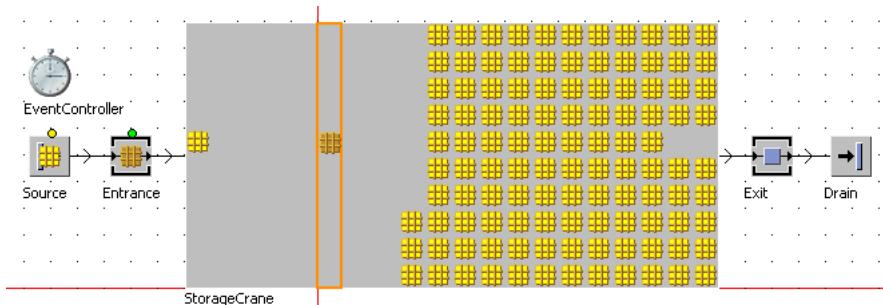
If you connect an object as entrance with the StorageCrane, then the StorageCrane looks for an empty place near the exit and moves the part to this storage place. As an automatic strategy to remove MUs, the StorageCrane offers

removal after a certain time (tab removal, Fig. 8.76). The swap takes place with a higher priority than the storage. Set in the example the removal time to five hours.



**Fig. 8.76 Removal Time**

If you choose "Shift blocking parts," then the crane automatically rearranges parts on top to get the underlying parts. Now, the StorageCrane already stores and removes MUs (Fig. 8.77).



**Fig. 8.77 StorageCrane**

#### 8.4.3.2 Customized Storage and Retrieval Strategies

Plant Simulation provides for the StorageCrane object attributes and methods (Table 8.6). So you can model more accurate storage and retrieval behavior.

**Table 8.6** Attributes and Methods of the StorageCrane

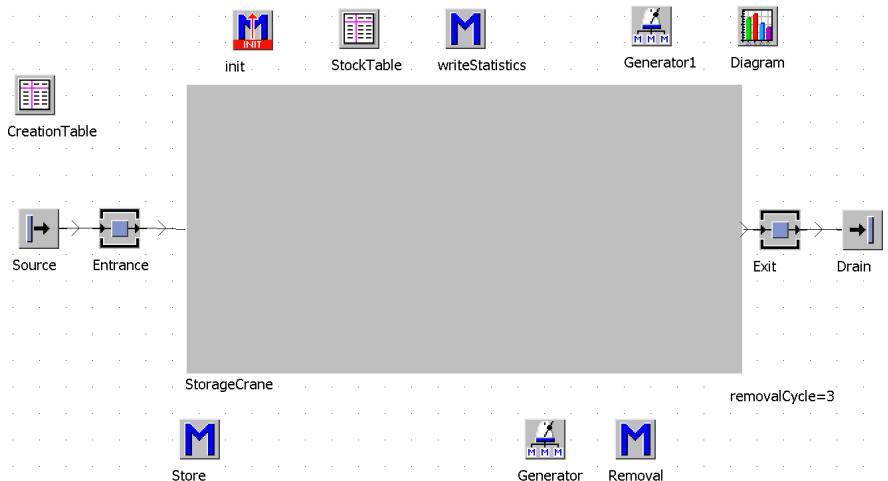
Attribute/Method	Description
<code>&lt;path&gt;.free(integer x, integer y, integer z)</code>	Sets the place at the position x, y, z on free (e.g. to undo a reservation).
<code>&lt;path&gt;.getPart(string part, integer number, integer priority)</code>	Requests a number of parts with the name part. The third parameter specifies the priority.
<code>&lt;path&gt;.getPartFromPosition(integer x, integer y, integer z, integer prio)</code>	Calls for a part at a specific x, y, z position.
<code>&lt;path&gt;.getPartFromPositionToObject (see help for parameters)</code>	Moves one part from the warehouse to an object in the area of the crane.
<code>&lt;path&gt;.getPartToObject(string name, integer number, integer priority, integer posX, integer posY, object destination, integer placeX, integer placeY)</code>	Moves a certain number of parts to a destination object.

**Table 8.6 (Continued)**

<path>.getStoreTable(object table)	Stores the inventory table of the warehouse in the given table.
<path>.getStoreXDim	Returns the x-dimension of the warehouse.
<path>.getStoreYDim	...
<path>.getStoreZDim	...
<path>.placeIsFree(integer x, integer y, integer z)	Checks whether the place at the position x, y, z is free.
<path>.reserve(integer x, integer y, integer z)	Reserves a storage place. At this storage place no automatic storage occurs.
<path>.shiftPart(integer prio, integer x1, integer y1, integer z1, integer x2, integer y2, integer z2)	Instructs the crane to move the part of x1, y1, z1 to x2, y2, z2.
<path>.storePart(object part, integer priority)	Moves the specified part to the entrance storage place of the warehouse (then automatic storage).
<path>.storePartFromObject (object obj, integer priority, integer x, integer y)	Loads a part from an object at the position x, y and stores it automatically.

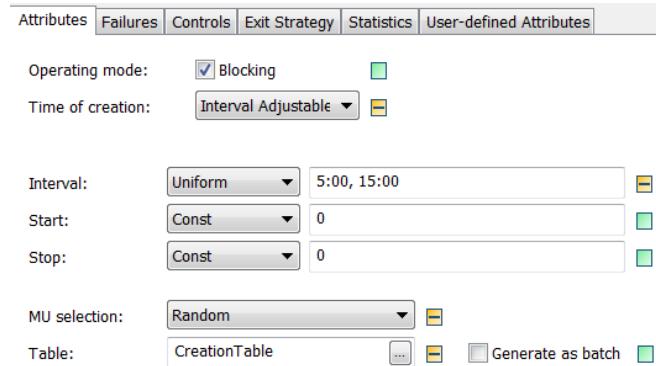
The following task is used for demonstration purposes:

In a warehouse, three kinds of parts are stored (Part1, Part2 and Part3). In the warehouse, the parts are assigned to specific areas in which they are stored. The removal is done according to requests individually. Create a frame in line with Fig. 8.78.

**Fig. 8.78** Example Frame

Create three entities (Part1, Part2 and Part3) and color them differently. The source randomly generates these three MUs at intervals of 10 minutes (+/- five

minutes) in equal proportions. The settings from Fig. 8.79 and Fig. 8.80 are necessary. Source:



**Fig. 8.79** Settings source

	object 1	real 2	integer 3	string 4
string	MU	Frequency	Number	Name
1	.MUs.Part1	0.33	1	
2	.MUs.Part2	0.33	1	
3	.MUs.Part3	0.33	1	

**Fig. 8.80** Creation Table

Ensure the following settings in the StorageCrane: Crane length: 21 m; crane width: 11 m; crane height: 4 m; portal position1: 0 m; portal position2: 11 m; storage places (x: 21, y: 11, z: 1); stack height: 1; entrance place: (1,6); exit place: (21,6); driving strategy: return to default position; default position (11,6).

### Pre-assignment of the Store

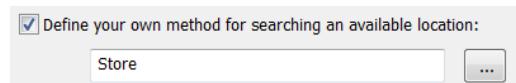
The store is to be initialized with five of each of the parts Part1, Part2 and Part3. Click on the button "Define Initialization of Store Area" in the tab Store. Enter in the table the occupation (Fig. 8.81). The position 1,1 is in the upper left corner.

	MU type	x	y	Number
1	.MUs.Part1	1	1	1
2	.MUs.Part1	1	2	1

**Fig. 8.81** Store Initialization

### Select Storage Place

By default, the StorageCrane stores the parts as close as possible to the exit place. To generate different behavior, you must determine the storage place via a method. Assign in the tab Strategy the method Store in the appropriate field (see Fig. 8.82).



**Fig. 8.82** User-defined Store Method

The store method must have the following signature:

```
(byref posX,posY,posZ:integer;be:object):Boolean
```

The return value of true indicates that you have found a free place. The position of the free place must be assigned to the transfer parameters. In the present case, Part1 is stored from x = 1 to x = 7, Part2 from x = 8 to 14 and Part3 from x = 15 to 21. When selecting the places, consider the entrance and exit points. Whether a place is occupied or not you can check using the method placeIsFree. This results, for example, in the following method:

```
(byref posX,posY,posZ:integer;be:object):Boolean
is
  i,k,start,finish:integer;
  placeFree:boolean;
do
  if be.name="Part1" then
    start:=1;
    finish:=7;
  elseif be.name="Part2" then
    start:=8;
    finish:=14;
  else
    start:=15;
    finish:=21;
  end;
  -- look for free place
  for i:=start to finish loop
    for k:=1 to 11 loop
      -- place 1,6 and 21,6 jump over
      if not((i=1 and k=6) or (i=21 and k=6)) then
        if storageCrane.placeIsfree(i,k,1) then
          placeFree:=true;
          posX:=i;
          posY:=k;
          posZ:=1;
        end;
      end;
    end;
  next;
next;
return placeFree;
end;
```

## Retrieval as Needed

The stored parts will be removed (settings tab Removal) after a certain time. If this does not correspond to reality, you must first disable automatic removal and then trigger the transfer via a method. In the example, each item of Part1, Part2 and Part3 should be removed at intervals of 10 minutes. The removal is scheduled to begin after one day of simulation time. Configure the generator so that it calls the method Removal after a start time of one day every 10 minutes. For the removal of the parts, you can use the method getPart. The method Removal might have the following content:

```

is
  part:string;
do
  if removalCycle=1 then
    part:="Part1";
  elseif removalCycle=2 then
    part:="Part2";
  else
    part:="Part3";
  end;
  StorageCrane.getPart(part,1,1);
  removalCycle:=removalCycle+1;
  if removalCycle > 3 then
    removalCycle:=1;
  end;
end;

```

### 8.4.3.3 Stock Statistics of the StorageCrane Object

The StorageCrane object itself provides a range of statistics. For the visualization of the stocks of stored MU types, however, you will need to set up a small evaluation. The StorageCrane object provides an inventory table. It contains the "storage coordinates" and a reference to the stored object (as string). With the method `<path>.getStoreTable(table)`, you can read the inventory table. As transfer parameters, you can pass a variable of type table, as well as a reference to a table object. Table objects are automatically formatted. Using the inventory table, you can create inventory statistics with little effort.

In the example above, create inventory statistics. It should include in the first column of a table the simulation time and in columns 2 to 4, the number of stored parts of Part1, Part2 and Part3 at this point in time. First, create a table according to Fig. 8.83.

	time 1	integer 2	integer 3	integer 4
string	Time	Part1	Part2	Part3
1				

Fig. 8.83 Stock Table

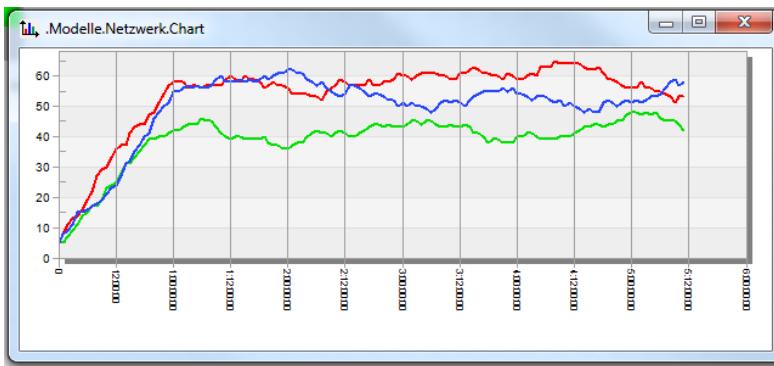
Set up the generator to call the method writeStatistics every hour. A method for counting the MUs in the store might look like this:

```

is
  stock:table[integer,integer,integer,string];
  part:string;
  number:integer;
  i,k:integer;
do
  stock.create;
  --load stockTables
  StorageCrane.getStoreTable(stock);
  -- the partnames are located columnwise in the head
  i:=2;
  stockTable[1,stockTable.yDim+1]:=ereignisverwalter.simTime;
  while(stockTable[i,0] /=""") loop
    part:=stockTable[i,0];
    number:=0;
    --count, 4th col
    for k:=1 to stock.yDim loop
      if str_to_obj(stock[4,k]).name=part then
        number:=number+1;
      end;
    next;
    stockTable[i,stockTable.yDim]:=number;
    i:=i+1;
  end;
end;

```

The result can be well-represented through an XY chart (Fig. 8.84).

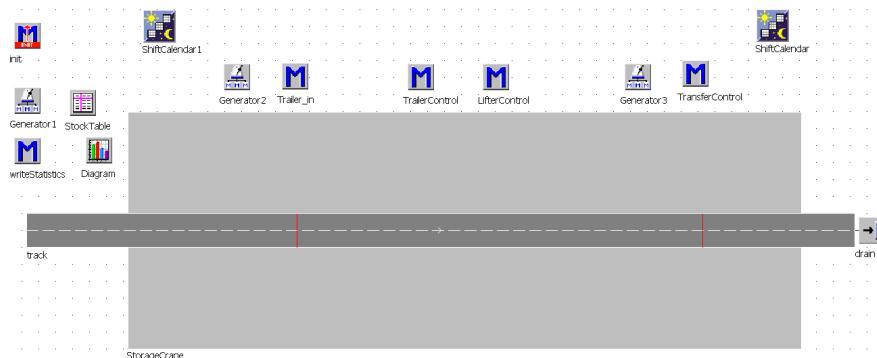


**Fig. 8.84** Stock Statistics

#### 8.4.3.4 Load and Unload the Store with a Transporter

Often, the store is loaded and unloaded by a transporter. In this case, the StorageCrane has to unload the transporter or move MUs to the transporter. The transporter must be located below the crane runway while loading. For storage, we need the method `<path> .storePartFromObject(...)`; for removal, we require the method `<path> .getPartToObject(...)`.

The following example should be modeled: A company gets its raw materials in big bags. They are delivered with the help of trailers (12 bags per trailer) and collected individually from the store by transporters. The complete handling of the big bags takes place via an overhead crane. The following conditions should be met: The delivery of big bags is done between 6:00 to 22:00 every two hours. Every 15 minutes (24 hours per day), a removal occurs. On weekends, the company does not work. The store has 120 storage places. For each storage place, an area of 3x3 meters is necessary. Create a frame like in Fig. 8.85.



**Fig. 8.85** Example Frame

Reproduce the following settings in the StorageCrane: crane length: 60 m; portal width: 21 m; portal height: 10 m; portal position 1: 0 m; portal position 2: 21 m; store x: 20, y: 7, z: 1; barred area from coordinate 2,4 to 19,4; entry place: 1,4; exit place: 20,4; driving strategy: remain at target; removal: deactivate all options. Create two transporters in the class library: trailer (capacity: 12; length: 15 meters) and forklift (capacity: one; length: three meters). Equip the transporters with an appropriate number of animation points. Create an entity BigBag. To model the delivery, one loaded trailer is generated on the track every two hours. It drives to a sensor, stops and enters there 12 jobs in the job list of the StorageCrane. The trailer waits until it is empty, then moves forward and is destroyed at the end of the track by the drain. For stock removal, a forklift is generated every 15 minutes at the end of the track. It drives backward to a second sensor and enters a removal order in the job list of the crane. The forklift waits until it is occupied and then moves forward to the end of the track, where it is destroyed by the drain.

## Delivery

The first step is the generation of the loaded trailer on the track at intervals of two hours. You will need a shift calendar, a generator and a method (shiftCalendar1, generator2, method Trailer\_in). Insert into the ShiftCalendar one shift from 06:00 to 22:00 (Monday to Friday) without any pauses. Set the generator to an interval of 15 minutes and assign the method Trailer\_in as interval control. The method Trailer\_in checks the current shift of the ShiftCalendar (they must not be ""), then generates a trailer and on the trailer big bags until the trailer is full. Therefore, the method Trailer\_in has the following content:

```
is
  trailer:object;
do
  if shiftCalendar1.getCurShift /= "" then
    trailer:=.MUs.Trailer.create(track,13);
    while not trailer.full loop
      .MUs.BigBag.create(trailer);
    end;
  end;
end;
```

Place a sensor on the track (so that the trailer is completely below the crane) and assign the method TrailerControl as the sensor control. A storage order can be generated with the method `storePartFromObject`. Pass the following transfer parameters:

- link to the MU
- priority
- crane position x
- crane position y

If the trailer is to be unloaded from back to front, then you have to start with the last MU (in this case `@.mu(12)`). The crane position X, you will need to adjust accordingly. The method for generating the storage orders could look like this:

```
(sensorID:integer;isBug:boolean)
is
do
  @.stopped:=true;
  --create store orders for the crane
  storageCrane.storePartFromObject(@.mu(12),1,1,4);
  storageCrane.storePartFromObject(@.mu(11),1,1,4);
  storageCrane.storePartFromObject(@.mu(10),1,2,4);
  storageCrane.storePartFromObject(@.mu(9),1,2,4);
  storageCrane.storePartFromObject(@.mu(8),1,3,4);
  storageCrane.storePartFromObject(@.mu(7),1,3,4);
  storageCrane.storePartFromObject(@.mu(6),1,4,4);
  storageCrane.storePartFromObject(@.mu(5),1,4,4);
  storageCrane.storePartFromObject(@.mu(4),1,5,4);
```

```

storageCrane.storePartFromObject(@.mu(3),1,5,4);
storageCrane.storePartFromObject(@.mu(2),1,6,4);
storageCrane.storePartFromObject(@.mu(1),1,6,4);
waituntil @.empty prio 1;
@.stopped:=false;
end;

```

### Removals

Every 15 minutes, a forklift is generated. It should move backward to a sensor at the exit of the store. For this, set the class of the forklift in the class library on backward. Place a sensor at the exit of the store on the track and assign the method lifterControl as the sensor control. For the generation of the forklift we need a shift calendar, a generator and a method (ShiftCalendar2, Generator3 and method transferControl). Insert into the ShiftCalendar one shift from 0:00 to 24:00 without any breaks (from Monday to Friday). With the help of the ShiftCalendar, we can easily determine the weekends (using the method getCurrShift, which returns an empty string for the unplanned period). Furthermore, it is useful to check the position of the trailer before the generation of the forklift. When the trailer leaves the warehouse (the front position is behind the "stopping sensor"), you need to wait to prevent collisions. The transferControl method, therefore, has the following content:

```

is
do
  if shiftCalendar.getCurrShift /= "" then
    if track.occupied then
      if track.cont.frontPos > 24.1 then
        -- trailer is moving outwards
        waituntil track.empty prio 1;
      end;
    end;
    .MUs.forklift.create(track,65);
  end;
end;

```

The removal takes place with the method getPartToObject. You need to pass the following parameters:

- Name of the MU, which is to be removed from the stock
- Number of MUs for the removal
- Priority
- Crane position X
- Crane position Y
- Object link to the destination object (forklift)
- Position X on the destination object (-1 for automatic selection)
- Position Y on the destination object (-1 for automatic selection)

In the current example, the forklift must stop at the sensor. It then asks for a big bag from the store and waits until it is loaded. The forklift starts moving forward

again, reaches the end of the track and is destroyed there by the drain. The method lifterControl has the following content:

```
(sensorID:integer;isBug:boolean)
is
do
  if @.name="Forklift" then
    @.stopped:=true;
    --request one part from the crane
    storageCrane.getPartToObject("BigBag",
    1,10,15,4,@,-1,-1);
    -- wait for loading
    waituntil @.numMu=1 prio 1;
    @.backwards:=false;
    @.stopped:=false;
  end;
end;
```

# Chapter 9

## Simulation of Workers

Plant Simulation has its own approach for the modeling of workers. It is based on the cooperation of Exporter and Importer. In some cases, you can also model workers with the help of transporters and tracks.

### 9.1 Exporter, Importer and Broker

#### 9.1.1 Function

The simplest simulation of workers and their services is the use of Exporter objects. You need three elements:

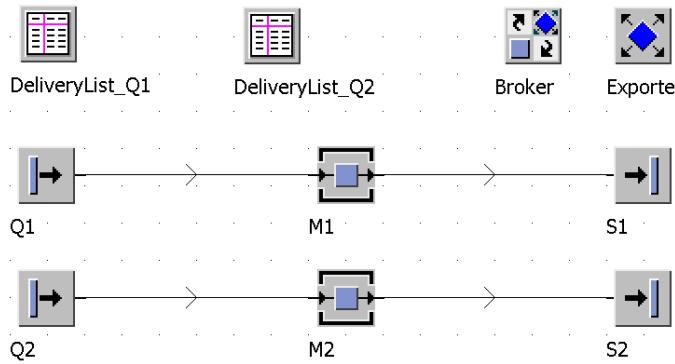
- Exporter: They offer certain services (e.g. for repair, operating, tool change, etc.) with a certain capacity.
- Importer: These request a certain amount of services (e.g. in cases of disturbances for repair, during the set-up for the machine, etc.).
- Broker: It brokers services between Exporter and Importer.
- Plant Simulation creates a range of statistics for the evaluation of service requests (e.g. waiting times) and service execution (e.g. number of required services).

#### Example: Exporter

For the setup and repair of two machines, only one resource is available. You should check (roughly), whether this is sufficient or not. Create a frame like in Fig. 9.1.

Create in the class library four entities (Part1, Part2, Part3 and Part4). Input the following settings:

Q1 blocking, sequence cyclically—DeliveryList\_Q1, Part1 50, Part2 25; Q2: blocking, sequence cyclically—DeliveryList\_Q2, Part3 33, Part4 33; M1: processing time: 20 minutes, set-up time: two hours, 75 per cent availability, MTTR 45 minutes; M2: processing time: 17 minutes, setup time: 1.5 hours, availability: 75 per cent, one hour MTTR.



**Fig. 9.1** Example frame

The exporter does not need many settings. Set the capacity to one; in the services table, you can define services provided by the exporter. Select Fail services (Plant Simulation then interrupts the service, which is executed by the exporter during the disturbance of the Exporter) and enter the Broker in the field Broker. Set the Exporter to an availability of 90 per cent with an MTTR of 10 minutes. The standard service of Exporters is "StandardService." This service should be requested if M1 or M2 is in the state setup or failed. To request the service, you need the dialogs Importer and Failure importer of the individual stations.

### Setup Importer

If you want to import services only for setting up (and not for processing), take the following steps:

1. Activate the Importer (mark the checkbox Active)
2. Uncheck the checkbox Common resources in the tab Importer
3. Select the Processing option, disable the inheritance of the service table, open the service table and delete the line with the entry "StandardService."
4. Select the Broker.

The service table for setting up the default service should remain registered. Perform these settings for M1 and M2.

### Failure Importer

If the failure importer is activated, the services listed in the service list are requested during any failure. Activate the failure importer in M1 and M2 and assign the broker of the network to the importer. You can follow the function of the exporter based on the LED (green for active, red for failed).

### 9.1.2 Exporter and Broker Statistics

The Exporter provides statistical data about the performed services (Table 9.1).

**Table 9.1** Exporter statistics (selection)

Method/Attribute	Description
<exp>.statSumFreeCapacity	Occupation of the exporter
<exp>.statExporterOperationalPortion	Time portions of the exporter
<exp>.statExporterFailedPortion	
<exp>.statExporterPausedPortion	
<exp>.statExporterUnplannedPortion	
<exp>.statServicesWorkingPortion	Statistical values regarding the working time of the exporter
<exp>.statServicesRepairingPortion	
<exp>.statServicesFailedPortion	
<exp>.statServicesSetupPortion	
<exp>.statServicesWaitingPortion	
...	

The broker collects data on the services supplied. The main statistical values can be found in table 9.2.

**Table 9.2** Statistical values of the Broker

Attribute/ Method	Description
<path>.statOpenRequests	Number of requests that could not be satisfied immediately
<path>.statSatisfiedRequests	Number of requests that could be satisfied immediately
<path>.statMediationTime	The time the importer had to wait for the satisfaction of their requests, in sum or on average
<path>.statMediationTimeMU	
<path>.statStayTime	The time that the exporter remained at the importers (sum or average)
<path>.statStayTimeMU	

## 9.2 Worker

The worker is a further development of the Exporter concept. You can use the workers to simulate employees. The simulation of employees is interesting particularly in the following situations:

- Repairs, maintenance
- Machine operation (the machine cannot operate without employees)
- Employees as transporters (carry parts)

### 9.2.1 The Worker-WorkerPool-Workplace-FootPath Concept

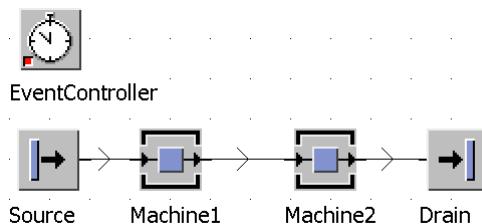
Workers are generated in the WorkerPool and remain there. The workers offer various services (e.g., repair, operate, and tool change). A broker mediates the workers to the individual workstations when they request the services. The broker sends the workers from the worker pool to the machines. If there is a footpath, then the worker walks from the WorkerPool to the machines along the footpath. The worker remains at his workplace while doing his job.

To simulate the workers, you need the following objects:

1. Broker
2. WorkerPool
3. Worker
4. FootPath
5. Workplace

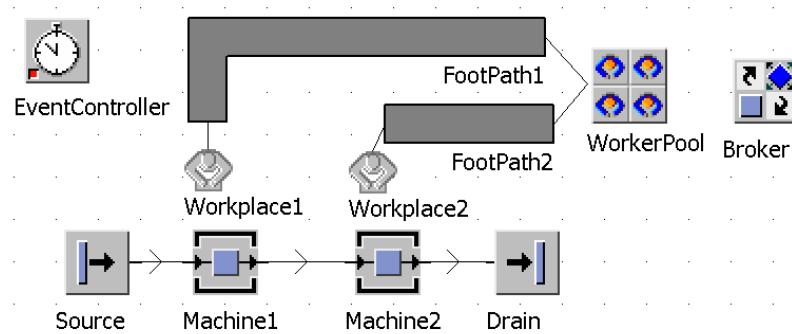
#### Example: Resources for repairs

The same employee maintains two machines. Both machines have a processing time of two minutes, an availability of 80 per cent, and an MTTR of 45 minutes. The staffroom of the service employee is 50 meters away from the machines. Create a frame like in Fig. 9.2.



**Fig. 9.2** Example frame

Complete the example. It is best to begin with the broker. Then, drag the WorkerPool into the frame so that it is placed close to the broker. Next, drag the workplaces into the frame. Drop it close to the machine and insert the footpaths into the frame. You can insert the footpaths using corner points, just like tracks and lines. Connect the WorkerPool with the footpath and the footpath with the workplaces (Fig. 9.3).



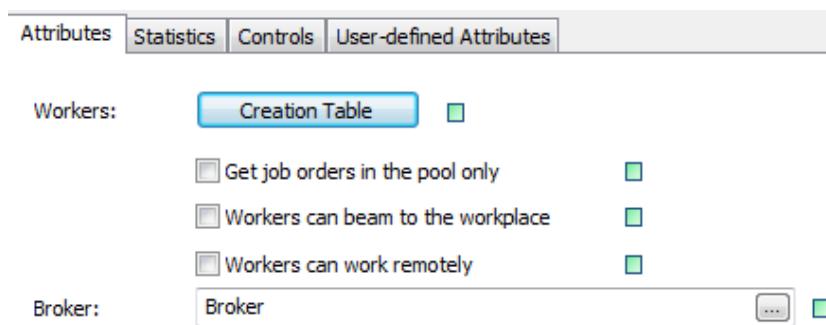
**Fig. 9.3** Example frame extension

### 9.2.2 *The Broker*

The broker mediates between the suppliers and demanders of services. You do not have to select settings in the broker. You have to propagate the machines as well as the resource pool to the broker.

### 9.2.3 *The WorkerPool*

The WorkerPool produces a number of workers according to a creation table and makes them available for the registered services after a request. If the workers do not work, Plant Simulation shows them on the WorkerPool. If you insert a WorkerPool close to a broker, the broker will be entered into the field Broker (Fig. 9.4).



**Fig. 9.4** WorkerPool

You have to enter the workers, which are managed in the pool, into the Creation Table. Click on Creation Table. Drag the workers to the Creation Table, and enter the amount (Fig. 9.5). In the column Additional Services, enter alternative services offered by the worker. The services are marked by a string. Hence, you can enter any of your needed services. To register more than one service, you can do it in the columns to the right next to Additional Services (maximum 30).

	Worker	Amount	Shift	Speed	Efficiency	Additional Services
1	*.Resources.Worker	1				repair

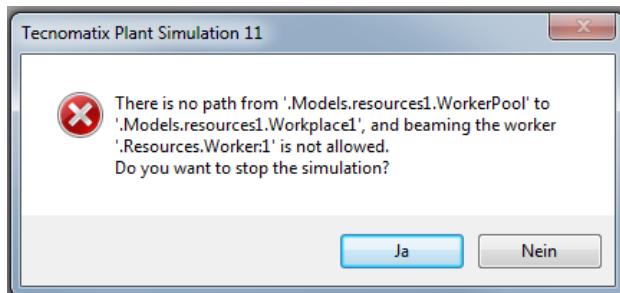
**Fig. 9.5** Worker creation table

Enter for the example, the service "repair."

You can also enter services directly in the dialog of the worker.

**Get job orders in the pool only:** If this option is selected, the worker must return to the WorkerPool between the individual orders (walk). If this option is cleared, then the worker can also receive an order on route to a job.

**Workers can beam to the workplace:** If this option is selected, the worker can walk directly to the workstation, even if there is no suitable footpath. If this option is cleared and there is no footpath to the workplace, Plant Simulation outputs an error (Fig. 9.6).



**Fig. 9.6** Error message

**Workers can work remotely:** If this option is selected, the worker can do his job, even if the respective workplace is already occupied (e.g. by another worker). Otherwise, you will get an error (Fig. 9.7).

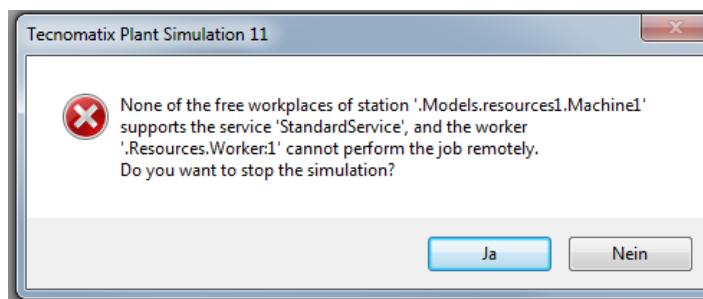


Fig. 9.7 Error message

**Broker:** Select the broker for the workplaces and the WorkerPool (you can drag the broker to the respective field of the WorkerPool).

#### 9.2.4 The Worker

You have to set worker-related settings in the class library because the instantiation of the worker is realized by the WorkerPool. The worker has a number of properties, which are important for the simulation (Fig. 9.8).

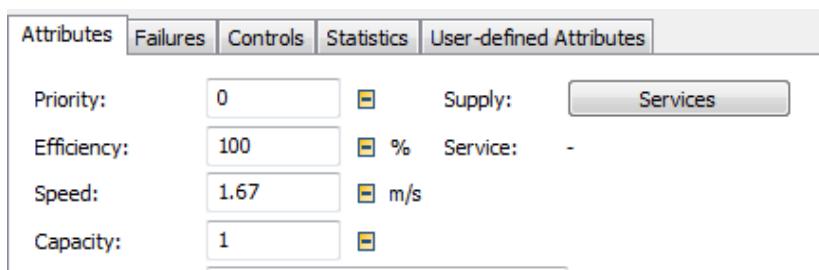


Fig. 9.8 Worker attributes

**Priority** (between 0 and 10): The higher the priority, the sooner a job will be performed

**Efficiency** (in percent): 50 per cent means that the worker needed twice the time for the job

**Speed:** Speed of the worker on the footpath

**Capacity:** Number of parts that the worker can carry at once

**Shift:** Name of the shift during which the worker works; if no shift is entered, the worker can work in all shifts

**Broker:** The broker will be assigned by the WorkerPool

Click the button Services to assign a range of services to the worker (Fig. 9.9).



**Fig. 9.9** Worker services

You can also fail the worker (analogous to the other blocks in Plant Simulation). The worker is not available for requests, while he is failed. If he is on a footpath, he returns to the WorkerPool and remains there until the failure is over.

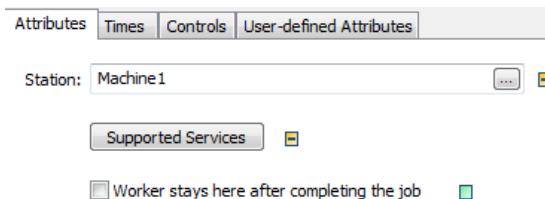
### 9.2.5 The Footpath

On the footpath, the workers move between workplaces and the WorkerPool. For that, they consume a time, resulting from the speed of the worker and the length of the footpath. You can disable the option Transfer Length in the tab Curve and enter the length manually, if necessary.

### 9.2.6 The Workplace

The worker stays on the workplace when he performs a job. Only one worker can stay on a workplace at any one time.

With the workplace, you connect an event of the machine (e.g. SingleProc) with a request for a service (e.g. failure: request of the service "repair"). The workplace needs to be assigned to another object, which happens automatically when you place the workplace close to another object. You can also select the respective object in the dialog of the workplace (Fig. 9.10).



**Fig. 9.10** Attributes of the workplace

Enter "repair" into the list Supported Services. Define here which activities can be executed at the workplace. The machine itself must request the service. You need to designate the broker and determine the service that will be required. Do this in the dialogs of the objects. The service will be requested if the machine fails. The Failure Importer is responsible for the request of the relevant services (tab Failure Importer, Fig. 9.11).

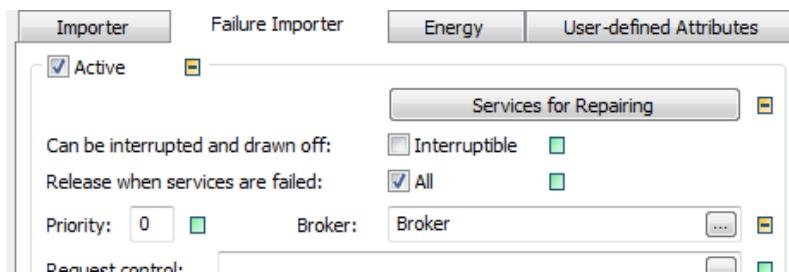


Fig. 9.11 Failure Importer

Activate the Failure Importer. Turn off inheritance for the services list (click on the green box next to the button Services for Repairing). Click the button Services for Repairing. Enter the service name ("repair") and the number of workers (one). Select the broker. Confirm your changes and run the simulation for a while. If a failure occurs, the worker moves from the WorkerPool to the machine and remains there until the failure is removed. Then he moves back into the WorkerPool (Fig. 9.12).

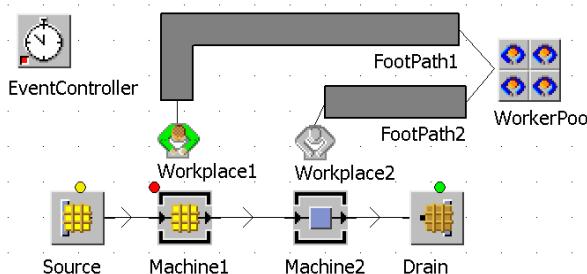


Fig. 9.12 Worker in action

### 9.2.7 Worker Transporting Parts

You can also use the worker for transporting parts. This is important, for example, if you need to simulate a multi-machine operation, in which workers have to transport the parts from one machine to the next. The active material flow objects SingleProc, ParallelProc, Buffer, PlaceBuffer, Sorter and Source have the exit strategy Carry Part Away.

#### Example: Resources Exit Strategy Carry Part Away

You should simulate a multi-machine operation. A worker takes parts from a buffer and carries them to Machine1. If Machine1 has finished, the worker carries the parts from Machine1 to Machine2 and then to another buffer. Create a frame like in Fig. 9.13.

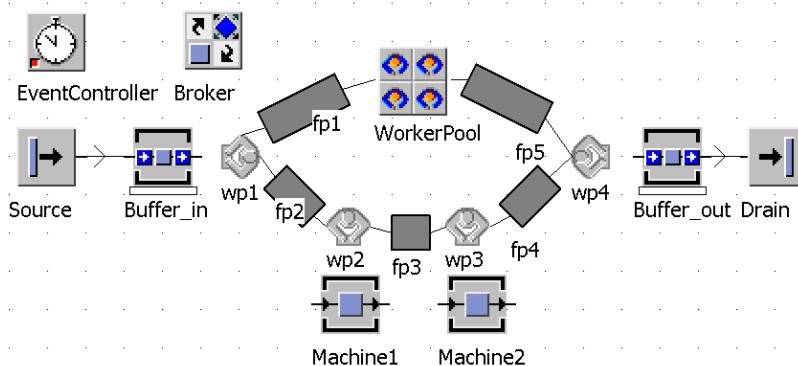


Fig. 9.13 Example frame

#### Settings:

Source: Interval: 1:30 minutes; Machine1, Machine2: 0:50 minutes processing time; buffer: capacity four parts each, 0:30 minutes processing time; worker: one worker; service: StandardService. Assign the Broker to the Machines and the WorkerPool. Assign the work places to the associated machines. The worker transports the part from one job to the next. Therefore, the next workplace must be assigned to each machine and each buffer. Open the dialog of Buffer\_in. Click the tab exit strategy, and select Carry Part Away from the drop-down list. Then click Apply (Fig. 9.14).

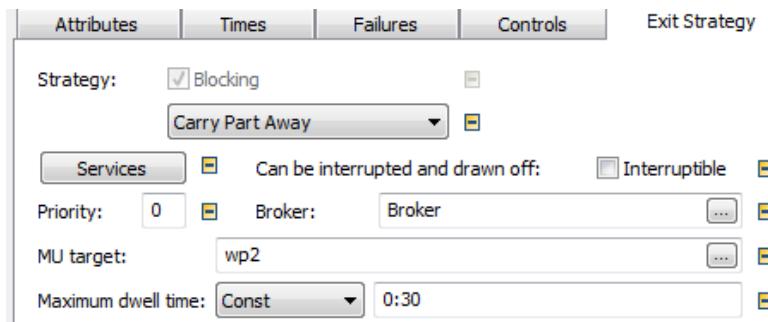


Fig. 9.14 Exit strategy

Select the broker. Specify the workplace that will handle the part next. Enter a maximum dwell time determining how long the worker must wait for a part from a machine (or the time that is necessary for the unloading/loading of the part). If you enter a longer dwell time than the processing time, the worker remains at the machine during processing and carries the finished part to the next station.

Do the same for Machine1 and Machine2. The worker now carries the parts from one machine to another. The worker does not check whether the succeeding

machine can receive the part. Therefore, the worker might carry a part to an already-occupied machine.

### 9.3 Worker Statistics

Analogous to the Exporter the worker collects a lot of statistical data. Plant Simulation provides values for all activities carried out by the worker, such as: statServicesWorkingPortion, statServicesRepairingPortion, statServicesFailedPortion, statServicesSetupPortion, statServicesTransportingPortion and so on. For simple analysis of the worker statistics, Plant Simulation provides a special library element, the Worker Chart. The WorkerChart is created as a user object and you must include it in your class library first. To do this, select File—Manage class library. You will find the Worker Chart in Libraries—Tools (Fig. 9.15).

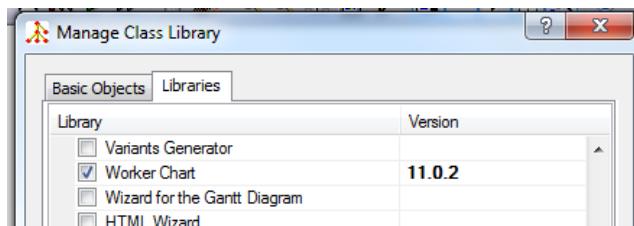


Fig. 9.15 Worker Chart

To create a Worker Chart, add a Worker Chart object into the frame and pull the WorkerPool to the Worker Chart.

### 9.4 Case Snippets for Worker Simulation

The use of the worker can be difficult to model in some cases. Here are some practical examples.

#### 9.4.1 *Loading of Multiple Machines by One Operator*

Plant Simulation allows many kinds of worker simulation. Often, the impact of human resources is on the line output of interest.

A worker can load several parts at the same time and feed different machines. Since Version 10, only very few adaptations of the basic behavior are necessary. The following simple example illustrates the approach. A worker collects three parts at the source and loads with these parts three machines. Create a frame according to Fig. 9.16.

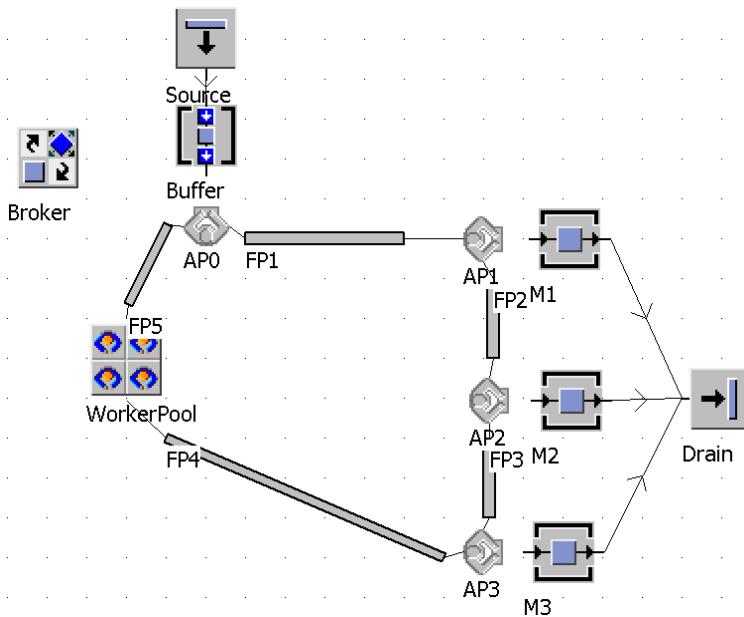


Fig. 9.16 Example frame

Ensure the following settings: The source produces parts at intervals of two minutes; the SingleProcs M1, M2 and M3 have a processing time of six minutes each. The worker has a capacity of three. The buffer has a capacity of four and the exit strategy Carry Part Away with the settings in Fig. 9.17.

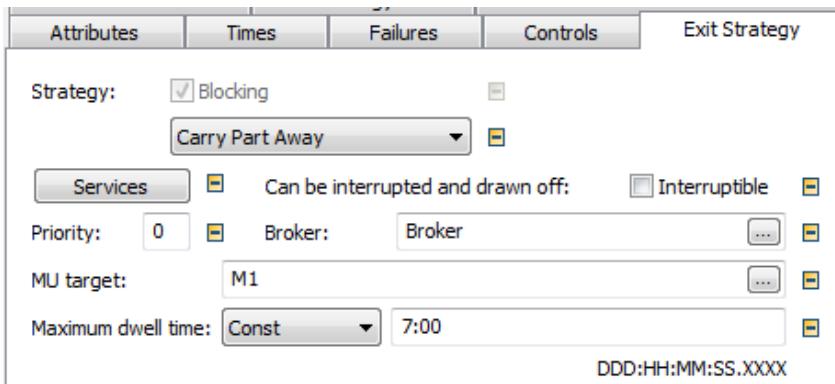


Fig. 9.17 Buffer exit strategy

The WorkerPool contains one worker. By default, the worker loads three parts (capacity) and unloads these at the station M1. In order to cause the worker to unload one part at each station, each part must have a different destination.

The exit strategy Carry Part Away stores the MU target in the attribute destination of the MUs. Starting with Version 10, the workplace has an input control and exit control. The exit control is called when the worker has left the workplace. This exit control can be used, for example, to re-address the MUs that carry the worker. Insert for this purpose an exit control in the workplace of the buffer (function key F4). In the present example, it is sufficient to address the second and third MU anew. Hence, the following small program is needed:

```
is
do
  @.mu(2).destination:=M2;
  @.mu(3).destination:=M3;
end;
```

Now, the worker loads M1, M2 and M3 with parts.

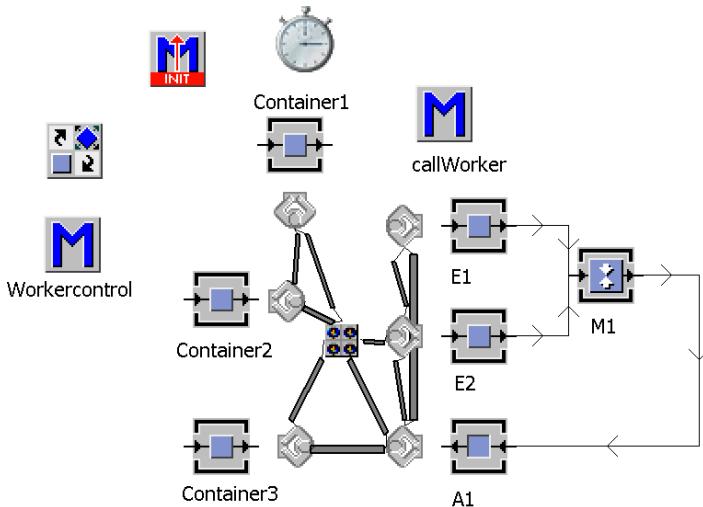
#### 9.4.2 *The Worker Loads and Unloads Containers*

The loading and unloading of containers by workers can be realized with the help of some lines of SimTalk. If you enter entrance controls in the workplaces, you can modify the default behavior of the worker (e.g. the loading in a container). To realize this, you need the SimTalk commands of Table 9.3.

**Table 9.3** SimTalk methods of the worker.

Method	Description
<worker>.goTo(destination:object)	Sends the worker to a particular station
<worker>.goToPool	Sends the worker back to the WorkerPool

You should simulate the following scenario: A worker takes two parts from different containers (Container1 and Container2) and places them at two different locations in an assembly station (E1 and E2). After assembly (M1), the worker removes the finished part (A1) and places it in a finished part container (Container3). Loading and unloading can be done parallel to the assembly process. Create a frame like in Fig. 9.18.



**Fig. 9.18** Example frame

Create three entities in the class library: Part1, Part2 and Assy. Set a capacity of 1,000 in the container. Change the capacity of the worker to two. Ensure the following settings: E1 and E2 processing time: zero seconds; M1 assembly mode: Delete MUs; exiting MU: New MU (Assy); assembly table: predecessors → Predecessor 2—one part, five minutes assembly time; workplace of E1: unloading time 45 seconds; workplace of E2: unloading time 30 seconds; workplace A1: loading time: 10 seconds; A1 exit strategy: Carry Part Away; MU target: Container3; maximum dwell time: 30 seconds (the worker should load only one part; therefore, he must not wait for a second part to fill his capacity). Rename the workplaces as follows:

Container1 → Ap1  
 Container2 → Ap2  
 E1 → Ap3  
 E2 → Ap4  
 A1 → Ap5  
 Container3 → Ap6

### Create Container

The subject of the example is not the transport of the container. The init method creates one container each on Container1, Container2 and Container3. The container on Container1 and Container2 will be filled with parts. At the end of the init method, the method callWorker is called to initiate the loading of the first part. The init method can appear as follows:

```

is
  container:object;
do

```

```
--create containers
container:=.MUs.container.create(container1);
while not container.full loop
    .MUs.Part1.create(container);
end;
container:=.MUs.Container.create(container2);
while not container.full loop
    .MUs.Part2.create(container);
end;
container:=.MUs.Container.create(container3);
callWorker;
end;
```

### Worker Call

You need to call the worker in two cases:

- if the loading points of the assembly station are empty (and at the beginning of the simulation)
- if the place A1 is occupied

The worker call for unloading station A1 works automatically through the exit strategy Carry Part Away. The call in case of empty station E1 must be programmed. Add a method callWorker into the station E1 as exit control (rear). The method callWorker waits until the worker is staying on the WorkerPool and then sends the worker to Container1 (goTo method), to take the first part.

```
is
    worker:object;
do
    waituntil workerPool.occupied prio 1;
    worker:=workerPool.contentsList[1,1];
    worker.goTo(container1);
end;
```

### Take parts from and place parts in containers

For loading from or into containers, adjust the default behavior of the worker. Add the method workerControl as entrance control in the workplaces in front of the containers. When loading from a container, the part is addressed (attribute destination) and transferred to the worker (the worker is addressed using @). The unloading of the parts to the station E1 and E2 is done automatically. The worker control for loading and unloading looks like the following:

```
is
    part:object;
do
    if ?=Ap1 then
        -- take one part from the container
        if container1.cont.empty then
            messagebox("Container1 empty!",1,13);
```

```

    eventController.stop;
end;
part:=container1.cont.cont;
part.destination:=E1;
part.move(@);
@.goTo(container2);
elseif ?=Ap2 then
  -- take one part from the container
  part:=Container2.cont.cont;
  part.zielort:=E2;
  part.move(@);
elseif ?=Ap6 then
  --place one part in container
  @.cont.move(container3.cont);
  @.goToPool;
end;
end;

```

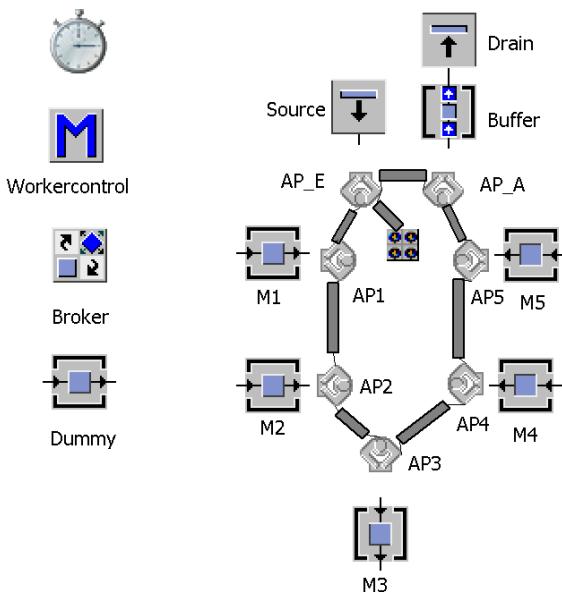
#### 9.4.3 Chaku-Chaku

Chaku-Chaku (Japanese for load-load) describes a system in which the material transport is realized within a line for the most part or entirely by workers. You can find an example in the collection of examples of Plant Simulation (Resources—Workers—Chaku-Chaku). The following example represents a Chaku-Chaku system without buffering between the machines. The worker thereby takes a part from the machine and loads the machine with a new part (without intermediate buffers). He transports the finished part to the next machine. In the example, the special methods and attributes of the workplace from Table 9.4 are used:

**Table 9.4** Methods and attributes of the Workplace and the SingleProc.

Method/Attribute	Description
<path>.station	Returns the station assigned to the workplace
<path>.resWorking	Returns true if the station is processing (observable)
<path>.MUTarget	Returns the MU target entered in the exit strategy Carry Part Away
<path>.unloadingTime	Reads the times that are entered in the workplace
<path>.loadingTime	

Create a frame according to Fig. 9.19.



**Fig. 9.19** Example frame

Ensure the following settings:

Worker capacity: two; source interval: 2:30 minutes, non-blocking; exit strategy: Carry Part Away; priority: 5; MU target: M1; maximum dwell time: zero seconds; M1, M2, M3, M4, M5 each ProcTime: 2:30 minutes; exit strategy: Carry Part Away; maximum dwell time: 10 seconds; MU target: next station; increase the priority for each station, the highest priority has M5; Ap1 to Ap5: unloading time: 10 seconds; entrance control: “workercontrol”; AP\_E and AP\_A: unloading time: five seconds.

If the station is occupied, a direct loading and unloading of the stations fails, because Plant Simulation checks the available capacity before unloading the part from the worker.

### Modeling Approach

If the worker is loaded with a part and the relevant station is occupied, then the worker must first wait until the processing is completed on the machine. Thereafter, the machine is unloaded. Unfortunately, the worker starts running immediately when the part is rearranged to him. Therefore, the part is first addressed and moved to a dummy place. To take into account the loading time on the machine, the machine will be blocked for the duration of loading and unloading. If the loading time is over, the part of the worker is loaded on the machine and the temporarily stored part from the dummy station is loaded on the worker. The worker then goes independently to the next station (value of the destination attribute of the part). The access to the machines is done via the workplaces with `?station`.

The worker control could have the following content:

```

is
  station:object;
  part:object;
do
  if @.occupied then
    part:=@.cont;
    station:=?.station;
    if station.occupied then
      waituntil station.resWorking = false prio 1;
      station.cont.destination:=station.MUTarget;
      station.cont.move(dummy);
    end;
    --simulate loading and unloading times
    station.startFailure(?/loadingTime+
                           ?/unloadingTime);
    waituntil station.operational prio 1;
    part.move(station);
    if dummy.occupied then
      dummy.cont.move(@);
    end;
  end;
end;

```

#### 9.4.4 Troubleshooting Depending on the Nature of the Failure

You can define failures in the most material flow elements. For more complex failure behavior, you can create failure profiles consisting of a series of individual failures. In many companies, there are the requirements to simulate different types of workers for troubleshooting different types of failures (e.g. one worker for mechanical failure, another worker for electrical failures). The problem is that the troubleshooting is permanently assigned to a particular service (tab Failure Importer—Services for Repairing, see Fig. 9.20).

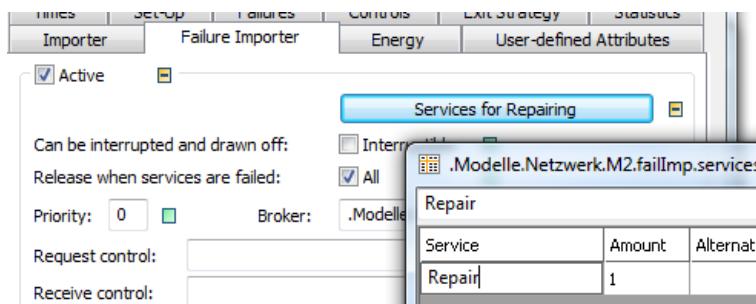
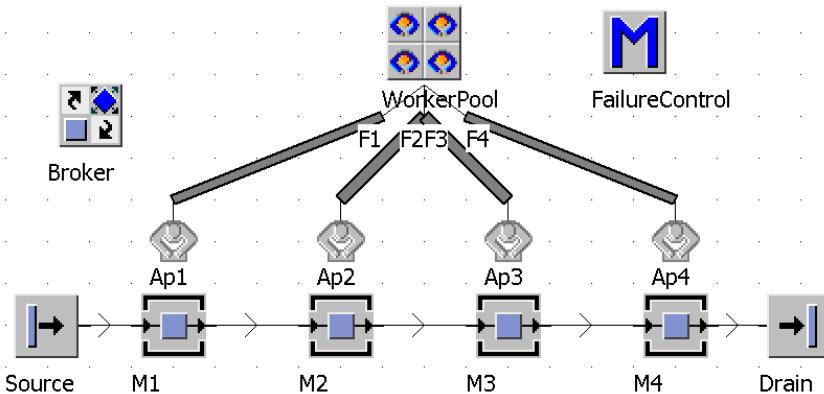


Fig. 9.20 Failure Importer

### Modeling Approach—Reset Service Table

One approach is to determine the name of the active failure profile when a failure occurs and, depending on the failure profile, to change the service list of the importer (Method `<path>.setServices`). According to the service then the right worker is requested. Create a frame like in Fig. 9.21.



**Fig. 9.21** Example frame

Duplicate twice the worker in the class library (Service\_engineer—Service: repair and Electrician—Service: electric). Create in the WorkerPool two service\_engineers and one electrician. The source generates parts in one-minute intervals. All SingleProcs have a processing time of one minute. All machines have the same two failure profiles:

- E\_failure: 95 per cent availability, 25 minutes MTTR, based on the operating time
- Tool\_problem: Interval uniformly distributed between one and two hours, failure duration between 10 and 30 minutes, uniformly distributed; based on processing time

For setting of the services, you can use the default failure control of the SingleProcs (Tools—Select controls). The failure control is called at the beginning and at the end of a failure event. Thereby are passed as arguments a Boolean value (disturbed) and the name of the active failure profile. Enter the method FailureControl as failure control into all SingleProcs. The method FailureControl could have the following content:

```
(failure: boolean; failureProfile: string)
is
  services:table[string,integer,string];
do
  services.create;
  if failure then
```

```

if failureProfile="E_failure" then
    services[1,1]:="electric";
    services[2,1]:=1;
elseif failureProfile = "Tool_problem" then
    services[1,1]:="repair";
    services[2,1]:=1;
end;
?.failImp.setServices(services);
else
    services[1,1]:="Nothing";
    services[2,1]:=1;
    ?.failImp.setServices(services);
end;
end;

```

If the failure is cleared (calling the failure method with failure = false), then an invalid service is registered. This means that the Failure Importer has to wait first (for the invalid service) and gives the FailureControl a chance to run.

#### 9.4.5 Collaborative Work of Several Workers

Plant Simulation requests the number of workers from the WorkerPool that you specify in the service table. In some cases, it is necessary to adjust the number of requested workers dynamically to the current available number of workers (even if more workplaces are to be served). Related to this is usually also the requirement to adjust the processing time of the processing station to the number of available workers. To realize this, several steps are required:

- Determination of the number of "free" workers
- Adjustment of the Services table
- Calculation and setting of the processing time of the processing station

To demonstrate this, the following scenario is intended:

A pool of five employees serves two machines (M1 and M2). To carry out the work on the machine M1, two employees are absolutely necessary. At the machine M2, a variable number of workers can operate (based on the workload of two hours, the required process time is reduced proportional to the number of workers). Create a frame based on Fig. 9.22.

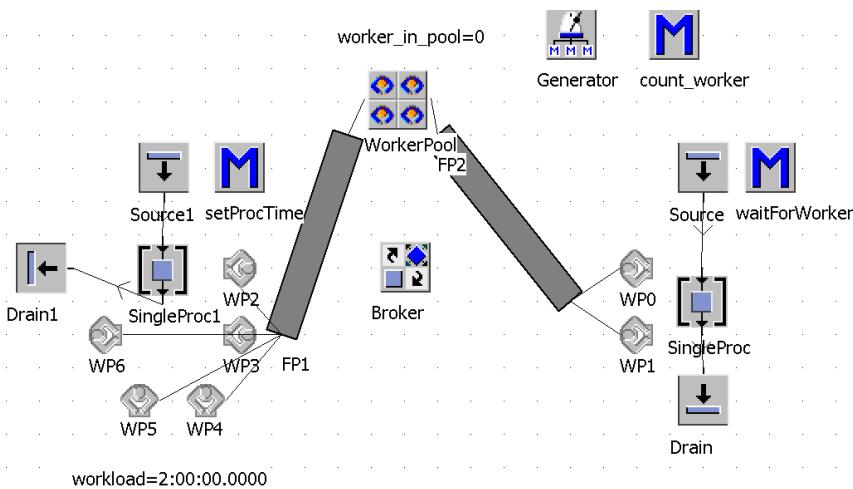


Fig. 9.22 Example frame

Create the following settings: Source1: Interval uniformly distributed between 15 minutes and two hours; exit control front: setProcTime; source: interval constant two hours; SingleProc: processing time: one hour. The generator calls the method count\_worker in intervals of five seconds. Create five workers in the WorkerPool. Activate the importer in SingleProc, assign the broker and set an amount of two units in the Service table (Fig. 9.23). The service is not interruptible.

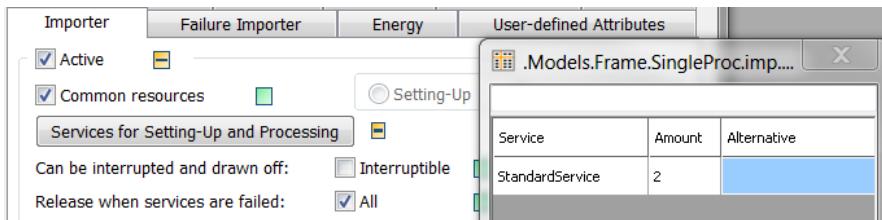


Fig. 9.23 Importer setting

Activate also in SingleProc2, the Importer and assign the Broker. Let the service table retain the default settings. The allocation of the available number of workers for SingleProc1 must be done in two steps:

1. Determination of the available number of employees
2. Insert the number of workers in the Service table

### Determination of the Available Number of Workers

In the example, a worker is considered to be available when he is in the WorkerPool or on the way to the WorkerPool. One way to determine this provides the inheritance of the worker. The attribute <path>.location returns the

current location of the respective child object and the attribute `<path>.previousLocation` of the previous location. If the worker is on a footpath, and the last location is not the WorkerPool, then he is on his way back to the WorkerPool and can be scheduled for new work. The method `count_worker` is called by a generator at an interval of five seconds, and saves the number of available workers in the variable `worker_in_pool`. The programming might look like this:

```

is
  i:integer;
do
  worker_in_pool:=0;
  --calculate number of available workers
  for i:=1 to .Ressources.Worker.numChildren loop
    if .Ressources.Worker.childNo(i).location =
      workerpool or
      (.Ressources.Worker.childNo(i).location.class=
       .Ressources.FoothPath and
       .Ressources.Worker.childNo(i).previousLocation/=
       workerpool) then
      worker_in_pool:=worker_in_pool+1;
    end;
  next;
end;

```

Whereas SingleProc requires a fixed number of workers, you must set in each case the available number of workers for the request for SingleProc1 at the arrival of a new part. The service amount is set with the method `<path>.imp.setServices(<table>)`. The service table must contain three columns. The first column contains the name of the service; the second column contains the number of required resources. The third column can contain the name of an alternative service. If the number of available workers is fixed, the processing time can be set for SingleProc1. You can implement a priority for SingleProc via a waituntil statement (e.g. wait until the source is empty, which means SingleProc is loaded and has the necessary number of workers requested from the WorkerPool). The method `setProcTime` requires the following content:

```

is
  services:table[string,integer,string];
  number:integer;
do
  count_worker;
  waituntil SingleProc1.empty and worker_in_pool>0
    and Source.empty prio 1;
  wait(1);
  services.create;
  --set number of services

```

```

services[1,1]:="StandardService";
services[2,1]:=worker_in_pool;
SingleProc1.imp.setServices(services);
SingleProc1.procTime:=workload/worker_in_pool;
@.move;
end;

```

### 9.4.6 A Worker Executes Different Activities at One Station

Under certain circumstances, a worker must sequentially perform different work at a station (e.g. at different workplaces). It is not possible to enter all activities to be performed line by line into the service list of the station, since then the station requests all these services at once and is waiting until all services are available at the same time. Rather, the process time must be split into the individual processes and the services must be requested sequentially. A similar approach can also be chosen if a worker is required for only part of the total process time at a station. Create a simple frame according to Fig. 9.24.

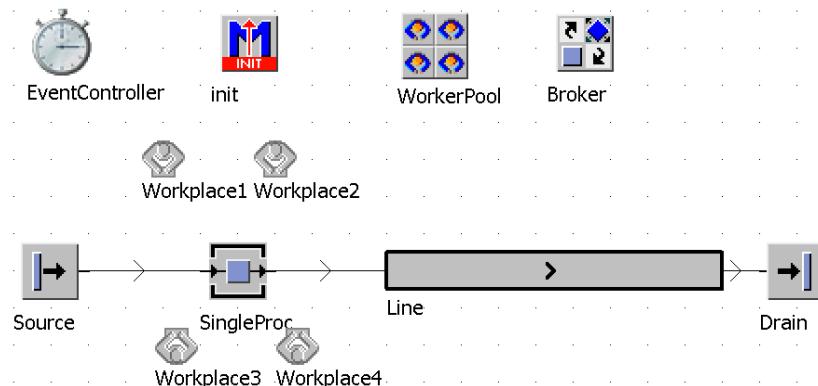


Fig. 9.24 Example frame

The source creates entities with an interval of four minutes. Assign to the worker in the class library the services A, B, C and D. Create in the WorkerPool exactly one worker. Activate the option “Workers can beam to the workplace”. Activate the importer in the SingleProc and assign the broker to the SingleProc.

#### Modeling Approach

You can model the process as sub-processes in which the MU is repeatedly relocated by the exit control of the SingleProc back to the same station. In order to prevent the entrance of a new MU on the SingleProc, you can lock the exit of its predecessor while processing the sub-processes. With the entrance control (before actions), you can set the respective processing time and the required service. If all sub-processes have been processed, the MU is relocated to the successor and the

exit of the predecessor is unlocked. The sub-processes can be set dynamically at the first call of the entrance control depending on the MU type. Create in the SingleProc an entrance control (front, before actions) and an exit control. Further, create in the SingleProc a user-defined attribute `cyclePos` (integer) and `cycle` (table). Format the table like in Fig. 9.25 and enter the data.

	string 1	time 2
string	Cycle Step	ProcTime
1	A	1:00.0000
2	B	1:00.0000
3	C	1:00.0000
4	D	1:00.0000

**Fig. 9.25** Table cycle

The init method sets `cyclePos` to 1 and unlocks the exit of the source:

```
is
do
  singleProc.cyclePos:=1;
  source.exitLocked:=false;
end;
```

The entrance control sets the processing time and the required service in the SingleProc according to the cycle table and the position within the sub-cycle:

```
is
  services:table[string,integer,string];
do
  --lock predecessor
  self.~.pred.exitLocked:=true;
  --load the right service
  services.create;
  services[1,1]:=self.~.cycle[1,self.~.cyclePos];
  services[2,1]:=1;
  self.~.Imp.setServices(services);
  --set procTime
  self.~.procTime:= self.~.cycle[2,self.~.cyclePos];
  -- update cyclepos
  self.~.cyclePos:=self.~.cyclePos+1;
end;
```

The exit control checks whether the cycle is complete and moves the part either to the successor or to start a new sub-cycle to the station itself.

```

is
do
  if self.~.cyclePos > self.~.cycle.yDim then
    @.move;
    self.~.cyclePos:=1;
    self.~.pred.exitLocked:=false;
  else
    @.move(self.~);
  end;
end;

```

#### 9.4.7 The Worker Changes Speed Depending on the Load

If the worker has to move heavy loads, it is possibly necessary to change the speed for the worker after loading. This can easily be done using the exit control of the workplace. Create a model according to Fig. 9.26.

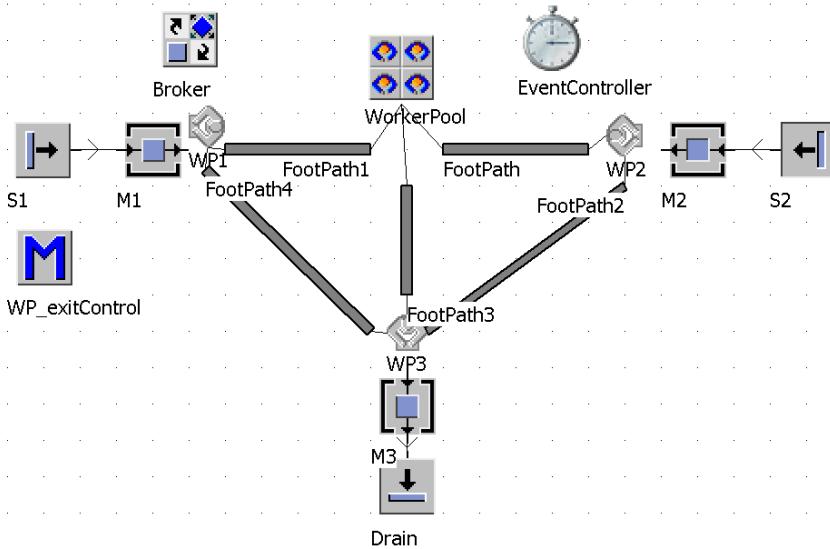


Fig. 9.26 Example frame

Ensure the following settings: The source produces MUs with a uniformly distributed interval between two and 10 minutes. M1 and M2 have a processing time of one minute and an exit strategy Carry Part Away with the destination M3. M3 has a processing time of 10 seconds. The WorkerPool contains one worker. Assign to all workplaces the method WP\_exitControl as exit control. With @ you can access the worker in the exit control of the workplace.

With `<worker>.occupied` you can determine whether the worker carries a part or not. Finally, you can change the speed of the worker with `<worker>.speed`. To switch the speed of the worker, you need two user-defined attributes in the worker class: `loadedSpeed` (speed, 0.25 m/s) and `emptySpeed` (speed, 1.5 m/s). The method `WP_exitControl` to change the speed depending on the load of the worker might look like this:

```
is
do
  if @.occupied then
    @.speed:=@.loadedSpeed;
  else
    @.speed:=@.emptySpeed;
  end;
end;
```

#### 9.4.8 Employees Working with Different Efficiency

A subject in the modeling of employees may be, for example, that they work in different shifts with different levels of efficiency (e.g. that the performance on the night shift is lower than in the other shifts). To take this into account in the simulation, you must adjust the efficiency of the worker, depending on the shift after the shift change. You can set the efficiency of the worker with the property `<worker>.efficiency`. Create a model like in Fig. 9.27.

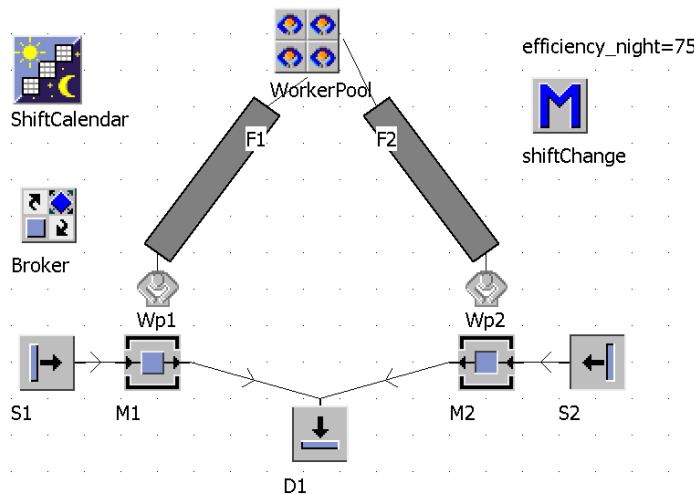


Fig. 9.27 Example frame

Ensure the following settings: M1 and M2 have a processing time of 10 minutes. Activate in M1 and M2 the importer (standard service for processing).

Create in the WorkerPool two workers per shift. Create in the ShiftCalendar three shifts like in Fig. 9.28. Between the shifts are 10 minutes of unscheduled time for the shift handover.

1	Shift-1	6:00	14:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	9:00-9:15; 12:00-12:45				
2	Shift-2	14:10	22:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	18:00-18:30; 20:30-21:00				
3	Shift-3	22:10	6:00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	00:00-00:30; 03:30-04:00				

**Fig. 9.28 ShiftCalendar**

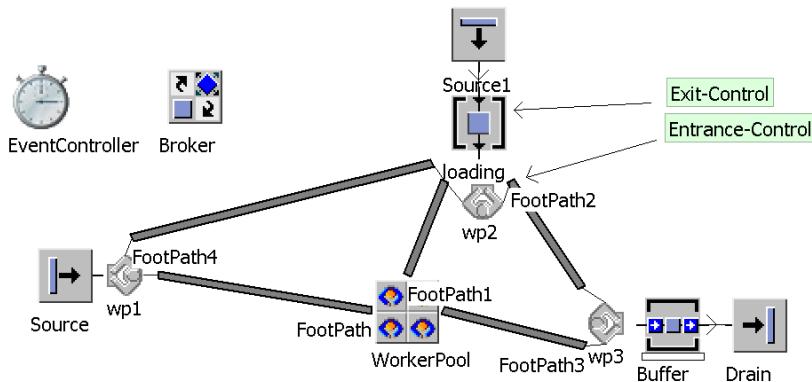
Assign the ShiftCalendar to the WorkerPool. In the ShiftCalendar, you can easily observe only the attributes pause and unplanned. These attributes can also be observed in the WorkerPool. In the constellation, with the shift handover time as unplanned time, you are able to use an observer to determine first the shift change and then to reduce the efficiency of the worker for the shift Shift-3. After the shift change to Shift 1, you reset the efficiency to 100 per cent.

Assign the method shiftChange as an observer for the attribute unplanned to the WorkerPool (Tools—Select Observer). The method ShiftChange would have the following content:

```
(Attribut: string; oldValue: any)
is
  actShift:string;
do
  if ?.unplanned=false then
    actShift:=shiftCalendar.getCurShift;
    if actShift="Shift-3" then
      .Ressources.Worker.efficiency:-
        efficiency_night;
    else
      .Ressources.Worker.efficiency:=100;
    end;
  end;
end;
```

#### 9.4.9 The Worker Loads Carriers and Transports Them

These cases can be realized with the exit strategy Carry Part Away. The only problem is the loading of the carrier by the worker. Hence, you will need the entrance control of the workplace. Create a simple frame like in Fig. 9.29.



**Fig. 9.29** Example frame

The worker should load four parts into a carrier and then transport the carrier to the buffer. Ensure the following settings: The source produces entities at intervals of one minute; the source1 produces containers every four minutes. The operator can carry only one part at the same time. Set the exit strategy Carry Part Away for the source (MU target: wp2) and the loading station MU target wp3).

## Preliminary Considerations

If the container on the place loading is ready for exit, then the worker is requested for transportation immediately. You must delay the request of the worker until the carrier is fully loaded. This works quite well with an exit control. Only with calling `@.move` is the worker requested. Create in the station loading an exit control (front). The exit control should have the following content:

```
is
do
  waituntil @.full prio 1;
  @.move;
end;
```

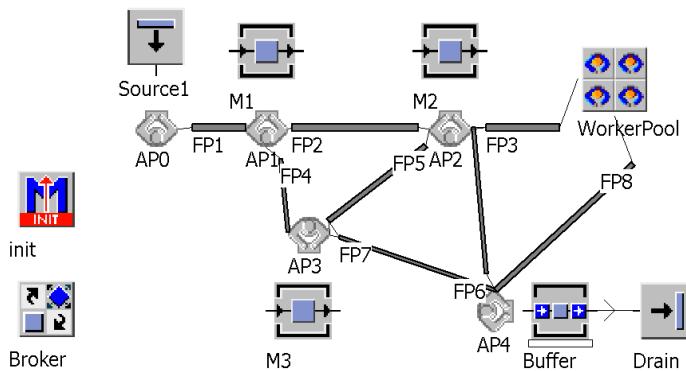
In the workplace wp2, you need an entrance control to unload the worker into the carrier. You must first wait until the carrier is available, then consider the time for unloading and move the part from the worker to the carrier. This yields the following entrance control:

```
is
do
  waituntil loading.occupied prio 1;
  wait(self.~.unloadingTime);
  @.cont.move(loading.cont);
end;
```

If the carrier is fully loaded, the worker transports it to wp3.

### 9.4.10 Multiple-Machine Operation

You are intended to simulate a multiple-machine operation. The operator places parts into a machine, does some work on the machine (e.g. clamping of parts). Then he starts the machining operation. The machine then processes automatically without the operator and the operator can load the next machine. The parts in the following example are to be processed sequentially on three machines. The worker can carry only one part. Create a frame according to Fig. 9.30. Duplicate in the ClassLibrary the SingleProc and set up the individual stations in the frame as instances of the duplicate.



**Fig. 9.30** Example frame

The WorkerPool creates one worker. The source has the exit strategy Carry Part Away with the MU target M1. In this example, the loading time of the worker should already appear as the working time of the machine in the statistics. The worker is also intended to be registered as working on the machine in this time.

#### Modeling Approach

To ensure that the worker remains at the machine after unloading the part, we need to divide the process into two parts:

- Insert (importer active) directly after moving the part to the machine
- Processing (importer not active + exit strategy Carry Part Away)

With the entrance control and the option Before Actions, you can make adjustments in the SingleProc that affect the current processing cycle. You can transfer the part at the end of the first sub-cycle to the same object and thereby change the settings for the second sub-cycle. The following adjustments must be set in the first sub-cycle (insertion of the part):

- Processing time for loading of the station
- Exit strategy: cyclic
- Importer active

At the end of the cycle (exit control), the part is moved to the same station. For the second sub-cycle (machine works without worker, at the end of the cycle the worker is requested for the transportation of the part), the following settings are required:

- Processing time for the automatic cycle
- Exit behavior Carry Part Away
- Importer not active

For the exit strategy Carry Part Away, you have to do some adjustments. Therefore, some SimTalk commands are necessary (Table 9.5).

**Table 9.5** SimTalk methods and attributes

Method/attribute	Description
<path>.exitStrategy	Sets the exit strategy of the object
<path>.importerActive	Activates/deactivates the importer
<path>.transportImp.BrokerPath	Sets the path of the broker for the exit strategy Carry Part Away
<path>.MUTarget	Sets the target of the transport for the exit strategy Carry Part Away

Insert into the duplicated SingleProc in the class library the user-defined attributes according to Fig. 9.31.

Name	Value	Type	C.
cycle	1	integer	*
InsertTime	1:00.0000	time	*
nextStation	(?)	object	*
OnEntrance		method	*
OnExit		method	*
ProcessingTime	6:00.0000	time	*

**Fig. 9.31** User-defined attributes

The entrance control of M should appear as follows:

```
is
do
  if ?.cycle= 1 then
    --if cycle 1
    --exit strategy Cyclic
    -- procTime: InsertTime
    --importeractiv true
    self.~.ExitStrategy:="Cyclic";
    self.~.procTime:=self.~.insertTime;
```

```

    self.~.importerActive:=true;
    ?.cycle:=2;
else
    --cycle 2
    -- exit strategy carry part away
    --importer not active
    -- procTime: processingTime
    self.~.ExitStrategy:="Carry part away";
    self.~.transportImp.BrokerPath:=root.broker;
    self.~.MUTarget:=self.~.nextStation;
    self.~.procTime:=self.~.processingTime;
    self.~.importerActive:=false;
    ?.cycle:=1;
end;
end;

```

The exit control has the following content:

```

is
do
    if self.~.cycle=2 then
        --start processing
        @.move(self.~);
    else
        --to the next station, if free
        waituntil self.~.nextStation.occupied=false
        prio 1;
        @.move;
    end;
end;

```

In each machine, you now need to set only the targets of transportation (user-defined attribute nextStation).

The init method sets all cycle values to one:

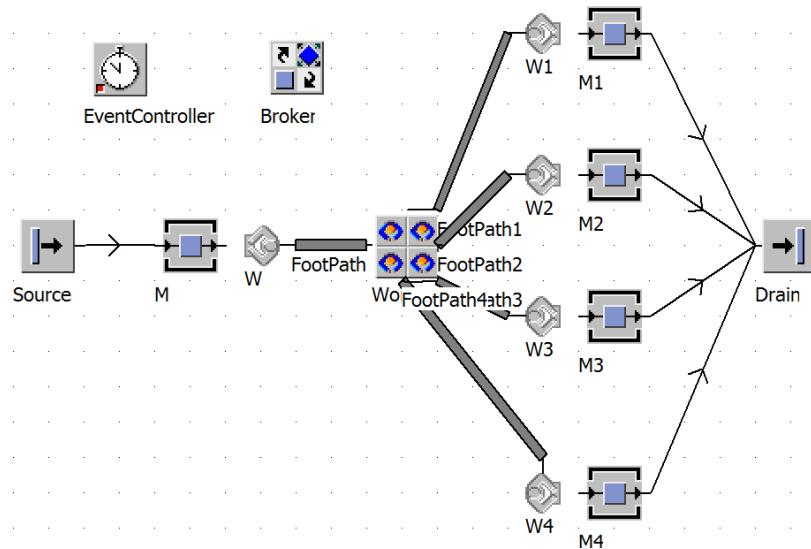
```

is
do
    M1.cycle:=1;
    M2.cycle:=1;
    M3.cycle:=1;
end;

```

#### 9.4.11 **Worker Loads Machines on Availability**

A worker should load four machines and should only go to available machines. Create a frame according to Fig. 9.32.



**Fig. 9.32** Example frame

Ensure the following settings: The source generates MUs every two minutes. All the stations have a processing time of two minutes. M1, M2, M3 and M4 have an availability of 50 per cent and an MTTR of 10 minutes. M has the exit strategy Carry Part Away. Enter the broker, but leave MU target empty.

### Modeling Approach

The destination of the transport through the worker is determined by the attribute destination of the MU, which the worker carries. If you use an exit control (front), you can set this value via SimTalk. In this example, you have to wait until one of the machines is able to process the part. Then, you set the attribute of the MU (@.destination) on the machine in question and call @.move. With @.move, the worker is requested from the broker. Enter an exit control (front) in the SingleProc M. The exit control should have the following content:

```

is
do
  waituntil (M1.operational and M1.empty) or
  (M2.operational and M2.empty) or
  (M3.operational and M3.empty) or
  (M4.operational and M4.empty) prio 1;
  --set destination for MU
  if M1.operational and M1.empty then
    @.destination:=M1;
  elseif M2.operational and M2.empty then
    @.destination:=M2;
  elseif M3.operational and M3.empty then
    @.destination:=M3;
  elseif M4.operational and M4.empty then
    @.destination:=M4;
  endif;
  @.move;
enddo;

```

```

elseif M4.operational and M4.empty then
  @.destination:=M4;
end;
--request worker
@.move;
end;

```

### 9.4.12 Worker Works with Priority (Broker Importer Request Control)

A station is to be given priority in the allocation of workers. You can achieve this through the allocation of a priority. On the other hand, you have the opportunity via the importer request of the Broker to intervene directly in the mediation process. Create a frame like in Fig. 9.33.

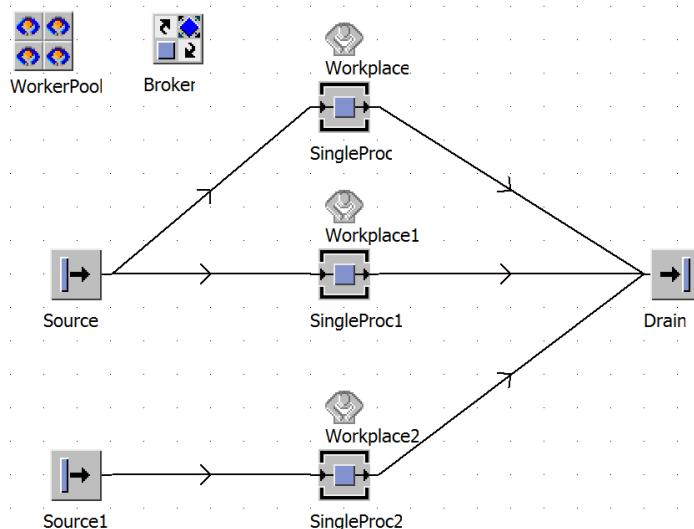


Fig. 9.33 Example frame

Prepare the following settings:

Source1: interval: 30 minutes; SingleProc and SingleProc1 Processing time: one minute; SingleProc: Processing time: five minutes. Activate in all SingleProcs the importer and assign the broker to it. Activate in the WorkerPool "Workers can beam to the workplace". The WorkerPool contains one worker. Create in the broker with F4 a method for the importer request. The importer request is called each time if an importer requests a worker or if there are unsatisfied importers and a worker reports as available. In the method, you need to allocate a suitable worker with `? .engage`. For this purpose, you can make a test export:

```
<broker>.testExportFor(<object>,<string>,<table>).
```

The broker tests whether the export of any of the available workers would be successful and returns a corresponding Boolean value. The successful exportable workers are stored in the third transfer parameter. This worker can be used for the mediation.

### Modeling Approach

In the importer request control is first searched whether SingleProc2 is among the unsatisfied importers. If so, then an attempt is made to mediate a worker. If SingleProc2 is not among the unsatisfied importers or the mediation was not successful, all other importers are satisfied. This yields the following control:

```
(obj : object;    -- Importer
  type : integer) -- Importer type
is
  workersToBeExported : table;
  testImportSuccessfull : boolean;
  unsatisfiedImporters:table;
  i:integer;
  obj1:object;
do
  workersToBeExported.create;
  unsatisfiedImporters.create;
  --unsatisfied importer
  ?.getOpenImporters(unsatisfiedImporters);
  --prefer singleProc2, otherwise first importer
  for i:=1 to unsatisfiedImporters.yDim loop
    if unsatisfiedImporters[1,i] = singleProc2 then
      obj1:=singleProc2;
      testImportSuccessfull := ?.testImportFor(obj1,
                                                unsatisfiedImporters[2,i],
                                                workersToBeExported);
      if testImportSuccessfull = true then
        ?.engage(obj1, unsatisfiedImporters[2,i],
                  workersToBeExported);
        return;
      end;
    end;
  end;
next;
--some other
for i:=1 to unsatisfiedImporters.yDim loop
  obj1:=unsatisfiedImporters[1,i];
  testImportSuccessfull := ?.testImportFor(obj1,
                                            unsatisfiedImporters[2,i], workersToBeExported);
  if testImportSuccessfull = true then
    ?.engage(obj1, unsatisfiedImporters[2,i],
              workersToBeExported);
  return;
end;
```

```
next;
end;
```

### 9.4.13 Determination of the Number of Workers with the ExperimentManager

If you want to change the number of workers with the ExperimentManager, then you have to change the creation table of the WorkerPool. A simple configuration could consist of a global variable and an observer in this variable for the attribute value. The global variable stores the number of workers. In the observer method, you read the creation table of the WorkerPool, set the number of workers according to the global variable and load the modified table back to the WorkerPool. Create a frame like in Fig. 9.34.

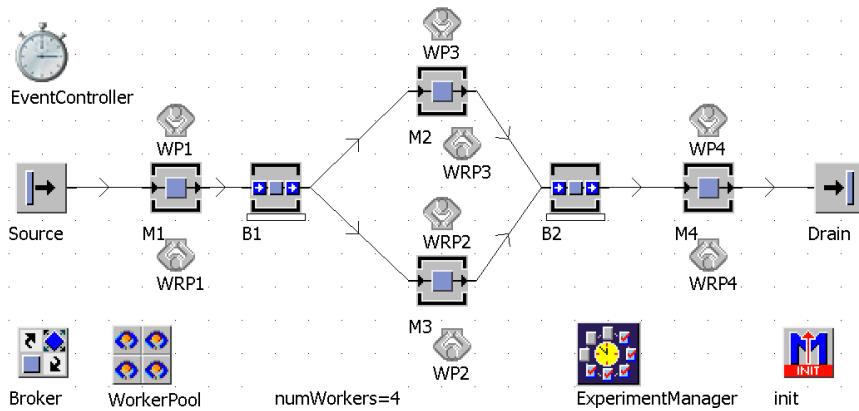


Fig. 9.34 Example frame

Ensure the following settings: M1 and M2 processing time: one minute; M3 and M4 processing time: two minutes; B1 and B2 have a capacity of 100. All SingleProcs have an availability of 75 per cent and an MTTR of 20 minutes. Activate in all SingleProcs the importer and the failure importer, and assign the broker. Create in the WorkerPool four workers. The ExperimentManager has Drain.statNumIn as a result value and changes the global variable numWorkers. Create experiments for numWorkers 4–10 with a step size of one. The observer method should have the following content:

```
(attribute:string; oldValue:any)
is
  rt:table;
do
  rt.create;
  --read the creation table
  workerpool.getCreationTable(rt);
```

```
-- set the new number
rt[2,1]:=numWorkers;
--re-assign the creation table to the WorkerPool
workerpool.setCreationTable(rt);
end;
```

## 9.5 Modeling of Workers with Transporter and Track

### 9.5.1 Modeling Approach

Especially for transportation tasks that must be performed by a worker (in the broader sense also for loading and unloading machines), the worker concept of Plant Simulation is somewhat bulky. If you need to simulate multi-machine operations, then you can easily realize this with the transporter. You can let the worker load parts (load to the transporter) and the worker can easily transport several parts. The worker can load the parts on the machine, wait for the completion and then unload the machine. Even operations that must be performed by the worker can be modeled easily. You have to consider the statistical information that is used to estimate the capacity of the worker. In general, it is necessary to record additional statistical information for the worker.

### 9.5.2 The Worker Follows a Process

Often, the work of the worker is defined as a sequence of steps. The worker must follow this process in the simulation. You can realize this by using a track and a set of sensors.

In the following example, simulate a multi-machine operation. The aim is to determine the utilization of the worker, including walking times, loading and unloading. Duplicate the track in the class library and rename the duplicate as worker\_track. Create in the worker\_track user-defined attributes according to Fig. 9.35.

Name	Wert	Typ
process	(?)	object
SensorControl		method
SensorList		table
step_var	(?)	object
virtPlace	(?)	object

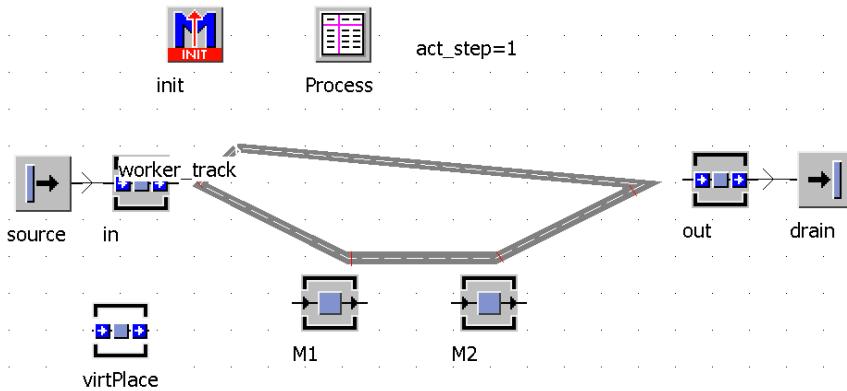
**Fig. 9.35** User-defined attributes

The first column of the sensorList must have the data type object. All objects used (including the buffer) must report the completion of the processing. Proceed as follows: Duplicate a SingleProc and add user-defined attributes as shown in Fig. 9.36.

Name	Wert	Typ
entranceControl		method
exitControl		method
processFinished	false	boolean

**Fig. 9.36** User-defined attributes

Assign to the entrance control and the exit control the corresponding methods. The entrance control sets the attribute processFinished to false and the exit control to true. Create a frame using the worker\_track and the adapted components as shown in Fig. 9.37.

**Fig. 9.37** Example frame

Insert into the worker\_track sensors at the positions of the stations and assign sensorControl as the method. Connect the end of the track with its beginning. Insert the stations into the sensorList of the track (corresponding to the sensor numbers, beginning with the buffer in). Set the attributes of the track as follows: process = process, step\_var = act\_step and virtPlace = virtPlace. Remove from the buffer out the exit control (if it exists). Enter according to Fig. 9.38 the process of the worker in the table process.

	string 1	string 2	time 3
string	Step	Code	ProcTime
1	load Part from Buffer in	unload	10.0000
2	go to M1	walk	
3	unload M1	unload	15.0000
4	load M1	load	10.0000
5	go to M2	walk	
6	unload M2	unload	5.0000
7	load M2	load	10.0000
8	go to out	walk	
9	move part to out	load	5.0000
10	go to in	walk	

**Fig. 9.38** Worker process

Duplicate a transporter and rename it in Worker. Replace the symbol of the transporter (you can insert the icon from the original Plant Simulation worker). Define two user-defined attributes in the new Worker: statLoadingTime (time) and statUnloadingTime (time). Both variables are used later for statistical analysis. Reduce the length of the worker to 0.1 meters. The worker is placed on the worker\_track via the init method. In the init method, the machines are pre-occupied with parts to ensure a smooth start of the simulation. Method init:

```
is
do
  .MUS.Worker.create(worker_track);
  .MUS.entity.create(M1);
  .MUS.entity.create(M2);
end;
```

The sensorControl checks at each sensor position the activity code of the process. If it is "unload," then that object must be finished before the part can be moved. If the worker is already transporting a part, then the part is temporarily placed on virtPlace. The object is unloaded. If "load" follows as the next process step, the part from virtPlace is relocated to the object. The processing times are considered with wait(xx) statements and the times are added to the user-defined statistics variables. Method sensorControl:

```
(SensorID : integer; IsFront : boolean)
is
  code:string;
  obj:object;
do
  --stop
```

```
@.stopped:=true;
-- read object from sensorList
obj:=self.~.sensorList[1,sensorID];
-- what to do ?
code:=self.~.process[2,self.~.step_var.value];
-- possible combination unload - load - walk
-- unload - walk
-- load - walk
if code = "unload" then
  -- if loaded , place part on virtPlace temporary
  if @.occupied then
    @.cont.move(self.~.virtPlace);
  end;
  waituntil obj.processFinished prio 1;
  obj.processFinished:=false;
  @.statUnloadingTime:=@.statUnloadingTime+
    self.~.process[3,self.~.step_var.value];
  wait(self.~.process[3,self.~.step_var.value]);
  obj.cont.move(@);
  self.~.step_var.value:=self.~.step_var.value+1;
code:=self.~.process[2,self.~.step_var.value];
if code="walk" then
  @.stopped:=false;
  self.~.step_var.value:=self.~.step_var.value+1;
else
  -- load and walk
  @.statLoadingTime:=@.statUnloadingTime+
    self.~.process[3,self.~.step_var.value];
  wait(self.~.process[3,self.~.step_var.value]);
  if self.~.virtPlace.occupied then
    self.~.virtPlace.cont.move(obj);
  end;
  self.~.step_var.value:=self.~.step_var.value+2;
  @.stopped:=false;
end;
elseif code = "load"  then
  -- place part and go away
  waituntil obj.empty and obj.operational prio 1;
  @.statLoadingTime:=@.statUnloadingTime+
    self.~.process[3,self.~.step_var.value];
  wait(self.~.process[3,self.~.step_var.value]);
  @.cont.move(obj);
  self.~.step_var.value:=self.~.step_var.value+2;
  @.stopped:=false;
end;
-- after finish the process start with step 1 again
if self.~.step_var.value > self.~.process.yDim then
```

```

    self.~.step_var.value:=1;
end;
end;

```

This concept can be expanded easily for other process elements, such as processing or checking.

### 9.5.3 The Worker Is Driving a Transporter

In the modeling of order-picking tasks or transportation, it may be necessary to determine the driving time of the worker. The worker of the Plant Simulation standard library cannot be located on a transporter. This also helps the use of the transporter in order to simulate the worker. You intend to simulate the following simple use case: A worker loads his transporter successively with four entities, then enters his car and drives to the unload position. There, he unloads the transporter part by part. Then, the worker enters the empty transporter and drives to the loading position. Create a simple frame according to Fig. 9.39.

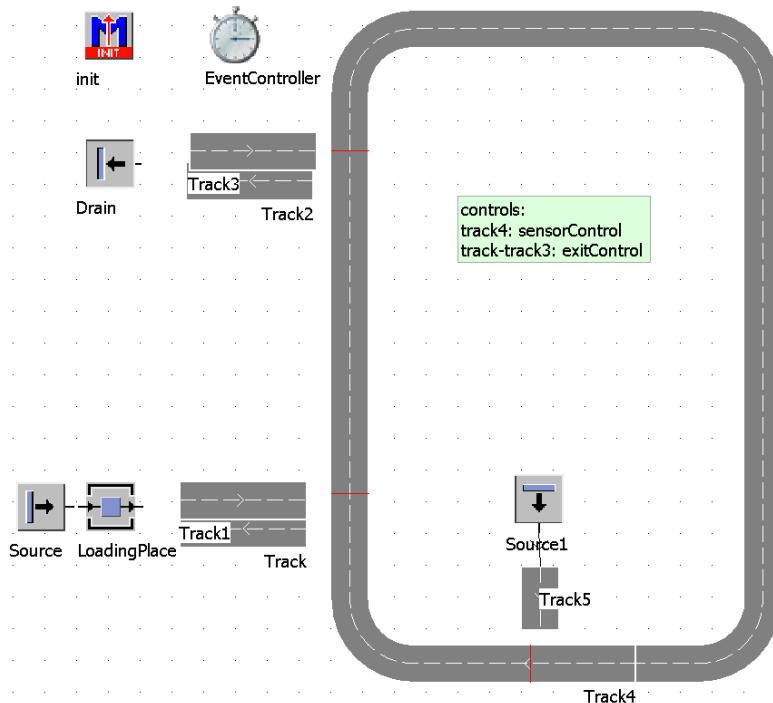
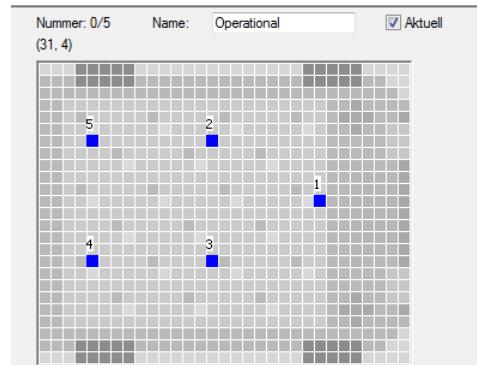


Fig. 9.39 Example frame

Source1 creates one worker. Duplicate for this a transporter and replace the symbol of the transporter by the worker icon. The default transporter needs a capacity of five (four entities and one worker). Create animation points on the transporter according to Fig. 9.40.



**Fig. 9.40** Animation points

Connect the end and the beginning of the track5 with a connector. Create on track5 three sensors (in the direction of travel starting at track5) and assign an internal method sensorControl. The init method creates one transporter in track4:

```
is
do
  .MUs.Transporter.create(track4, 2);
end;
```

### Place the worker on the transporter

After initialization, the worker is to be moved from track5 to the transporter on track4. You can solve this with the first sensor and the sensorControl on track4 and an exit control on track5. The sensorControl should contain the following: If the transporter is empty, stop it and wait until it is full. Then, restart the transporter.

```
(SensorID : integer; Front : boolean)
is
do
  if sensorID=1 and @.empty then
    @.stopped:=true;
    waituntil @.occupied prio 1;
    @.stopped:=false;
  end;
end;
```

At the end of track5, the worker is placed on the transporter (track4.inhalt).

```
is
do
  @.move(track4.cont);
end;
```

### The worker loads the transporter

The control for the loading of the transporter can be programmed into two parts:

- Transporter: The transporter stops at the sensor. The worker gets off (on the track). The transporter waits until it is fully loaded, and then starts again.
- Worker: The worker loads at the end of track one part each from the loading place. At the end of track1, the worker loads the part on the transporter. After loading the last part, the worker moves himself to the transporter. Before he can move himself to the transporter, he usually needs to empty his place (pe(1)).

To a) sensorControl for sensor 2:

```
elseif sensorID=2 then
  @.stopped:=true;
  @.cont.move(track);
  waituntil @.numMu=5 prio 1;
  @.stopped:=false;
```

To b) exit control track:

```
is
do
  loadingPlace.cont.move(@);
  @.move(track1);
end;
```

Exit control track 1:

```
is
do
  @.cont.move(track4.cont);
  if track4.cont.numMu = 4 then
    --clear place for worker
    if track4.cont.pe(1,1).cont /= void then
      track4.cont.pe(1,1).cont.move(
        track4.cont.pe(5,1));
    end;
    @.move(track4.cont);
  else
    @.move(track);
  end;
end;
```

### The worker unloads the transporter

Analogous to load, you can split the control into two areas:

- Transporter: The transporter stops, the worker gets off (on track3). First, you wait until the vehicle is completely emptied and then until the worker has entered the transporter. Then, you start the transporter.

- b) Worker: At the end of track3, he load one part or get on the transporter when it is empty. At the end of track2, he load the part in the drain.

To a) SensorControl for sensor id = 3

```
elseif sensorId=3 then
  @.stopped:=true;
  @.cont.move(track3,3.5);
  waituntil @.empty prio 1;
  waituntil @.numMu=1 prio 1;
  @.stopped:=false;
```

To b) Exit control track3: unload transporter, enter the transporter

```
is
do
  if track4.cont.occupied then
    track4.cont.cont.move(@);
    @.move(track2);
  else
    @.move(track4.cont);
  end;
end;
```

Exit control track2: move part to the drain and walk back

```
is
do
  @.cont.move(Drain);
  @.move(track3);
end;
```

# Chapter 10

## The Fluids Library

From Version 12 onward, Plant Simulation provides the Fluids library. With the help of fluid elements, it is possible to model liquids and bulk materials with relative ease. The library contains only a few elements. By combining these elements, however, you can map a variety of situations.

### 10.1 The Fluid Elements, Continuous Simulation

The Fluids library provides the following elements:

- Pipe
- Tank
- Mixer
- Portioner
- FluidSource and FluidDrain
- MaterialsTable

The MaterialsTable stores all information about the materials used. In the MaterialsTable, you can also define recipes. The Tank can hold a certain amount of material and output it at a constant rate. The Mixer generates, according to a recipe, an intermediate or a final product. You can define process times for the Mixer. The Portioner divides the fluid into portions and creates MUs (e.g. bottled in containers). FluidSource and FluidDrain represent the boundaries of the continuous simulation. All elements of the Fluids library are connected by pipes.

#### **Example: Fluids 1**

You are intended to simulate a part of a bottling plant. From juice concentrate and water, a juice is made and then bottled. Create a frame as in Fig. 10.1. The MaterialsTable contains all the information about the materials and recipes used. You can also assign colors in order to distinguish among the different materials better. In the example, juice is to be mixed using 10 per cent concentrate and 90 per cent water. Thus, you need three entries in the MaterialsTable (see Fig. 10.2).

In the FluidSources, specify which material is produced and what flow rate is applied to the simulation (e.g. pump power). Define a reference to your MaterialsTable for other required information. The source for the concentrate should have the settings shown in Fig. 10.3 for a flow rate of 1 l/s.

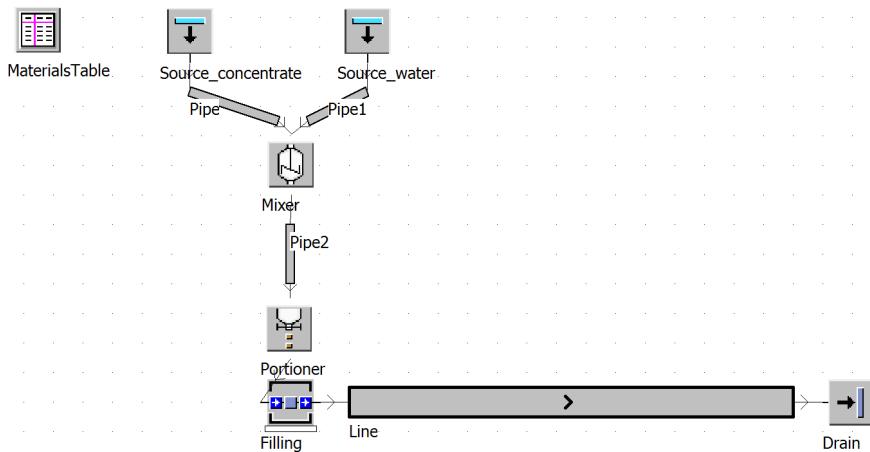


Fig. 10.1 Example frame

string 1	real 2	integer 3	real 4	string 5	string 6	real 7	string 8	string 9	real 10	st 11	
string	Material	Density [g/cm <sup>3</sup> ]	Color	Product	Amount	Unit	Ingredient 1	Amount	Unit	Ingredient 2	Amount
1	Water	1.000	16768582		1.000	l					
2	Concentrate	0.900	255		1.000	l					
3	Juice	0.980	33023		1.000	l	Water	0.900	l	Concentrate	0.100

Fig. 10.2 MaterialsTable

Attributes   Failures   Statistics   User-defined Attributes

Outflow rate:  l/s

Material:

Materials table:

Fig. 10.3 FluidSource

Settings for Source\_Water: Material: Water; Outflow rate: 1 l/s, MaterialsTable. The Mixer operates as follows:

The Mixer is filled with raw materials according to the recipe of the product (up to the capacity). The mixture is then processed (processing time). At the end of the process time, the mixer passes on the product at the set flow rate. If the mixer is empty (and the recovery time, e.g. for cleaning operations, is over), it is filled and the cycle begins again. For activities during the change of products, you can define a set-up time. Prepare settings as in Fig. 10.4 in the Mixer.

Attributes	Times	Failures	Statistics
Outflow rate:	0.375	<input type="button" value="l/s"/>	
Volume:	1000	<input type="button" value="l"/>	
Product:	Juice	<input type="button" value=""/>	
Product amount:	-1	<input type="button" value=""/>	<input type="button" value="l"/>
Materials table:	MaterialsTable	<input type="button" value="..."/>	<input type="button" value=""/>

**Fig. 10.4** Setting for Mixer

In the field "Product amount," set the amount of the product after the mixing process. If you enter -1, then the volume of the product after mixing remains the same.

The Portioner divides the product into MUs. For this purpose, you need to set the volume per MU and determine an MU that is to be produced. The time interval between the generations of the MUs is determined by the outflow rate of the predecessor and the volume of the MU. In this example, a bottle is to be filled at a rate of 0.375 l/sec with a volume of 0.7 l. The Portioner accordingly generates a bottle every 1,866 seconds. Create in the class library an entity named "Bottle." Set the width and length of the bottle to 15 cm. Adjust the speed of the line to 0.2 m/s. The buffer Filling has a capacity of one and no processing time. Prepare settings in the Portioner as in Fig. 10.5.

Attributes	Times	Failures	Controls	Exit Strategy
MU:	.MUs.Bottle	<input type="button" value="..."/>	<input type="button" value=""/>	<input type="button" value=""/>
Amount per MU:	0.75	<input type="button" value="l"/>	<input type="button" value=""/>	
Fluid from predecessor:	1	<input type="button" value=""/>	<input type="button" value=""/>	

**Fig. 10.5** Settings for Portioner

The fluid elements provide a range of statistical analysis, e.g. regarding the consumption of materials.

## 10.2 The Tank

The Tank is used for the temporary storage of raw materials and products. It has a capacity and an outflow rate. You can define a setup time to model cleaning. The material flow can be controlled through the opening and closing of pipes (similar to valves). You can open and close the pipes using the attribute `<pipe>.pipeOpened` (Boolean). The pipe itself has no volume in Plant Simulation.

### Example: Fluids—Multi-tank system

A system is filled periodically every two hours with 200 liters. To be able to bridge supply shortages, there are three tanks. You should fill the empty tanks. Removal takes place alternately (cyclically in turn) from the containers. Create a frame as in Fig. 10.6.

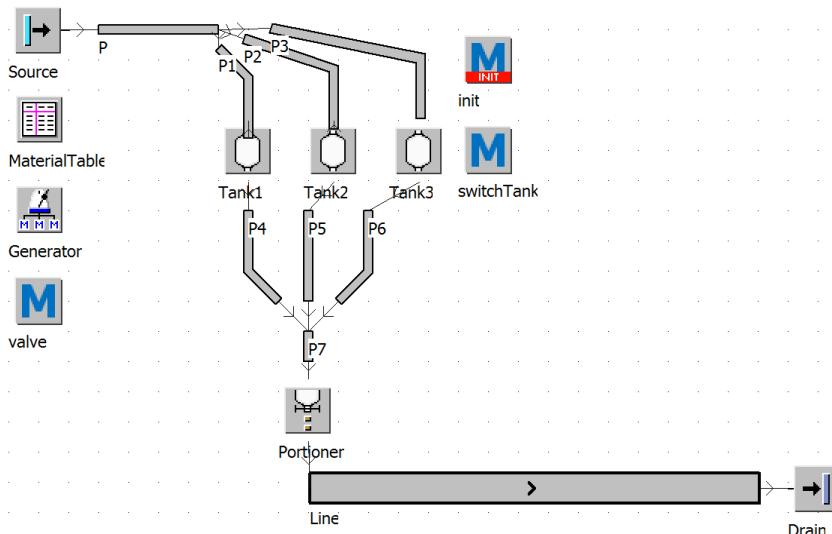


Fig. 10.6 Example frame

Ensure the following settings: Source: Standard settings; assign the MaterialsTable, Tank1, Tank2, Tank3: capacity 300 liters, outflow rate: 0.03 l/s; Portioner: MU Entity, 0.2 l per MU; Line: speed 0.4 m/sec.

### Initialization

The model is to be initialized. The source does not deliver at the beginning. All access pipes (p1, p2, p3) are closed. Tank1 and Tank2 should have an initial stock. P4 should be opened at the beginning (p5 and p6 closed). The pipes can be opened and closed using the attribute `<pipe>.pipeOpened`. The tanks are filled using the method `setCurrentContent (<amount>, <material>, <MaterialsTable>)`. The init method should look like this:

```

is
do
  --no delivery
  p.pipeOpened:=false;
  --all entrance pipes closed
  p1.pipeOpened:=false;
  p2.pipeOpened:=false;
  p3.pipeOpened:=false;
  --initial stock in tank1 and tank2
  tank1.setCurrentContent(200,
    "StandardMaterial",MaterialTable);
  tank2.setCurrentContent(200,
    "StandardMaterial",MaterialTable);
  --exit tank1 is open
  p4.pipeOpened:=true;
  p5.pipeOpened:=false;
  p6.pipeOpened:=false;
end;

```

### Delivery

For a delivery, you only need to open the pipes in question. To deliver 200 liters at a flow rate of 1 l/s, close the pipes after 200 seconds. With `<tank>.currentAmount`, you can query the level of the tanks to open the right "path" of pipes. The method valve is called every two hours from the generator. It should have the following content:

```

is
do
  if tank1.empty then
    p.pipeOpened:=true;
    --open pipe to tank1
    p1.pipeOpened:=true;
    p2.pipeOpened:=false;
    p3.pipeOpened:=false;
    wait(200);--200 liters
    --stop delivery
    p.pipeOpened:=false;
  end;
  if tank2.empty then
    p.pipeOpened:=true;
    p2.pipeOpened:=true;
    p1.pipeOpened:=false;
    p3.pipeOpened:=false;
    wait(200);--200 liters
    --stop delivery
    p.pipeOpened:=false;
  end;
  if tank3.empty then

```

```

p.pipeOpened:=true;
p3.pipeOpened:=true;
p2.pipeOpened:=false;
p1.pipeOpened:=false;
wait(200);--200 liters
--stop delivery
p.pipeOpened:=false;
end;
end;

```

### Switching Tanks

If a tank is empty, it should be switched to the next full tank. In the example, the switch should be fixed from Tank1 to Tank2, from Tank2 to Tank3 and from Tank3 to Tank1. The switch works again with the opening and closing of the respective pipes. You can use an observer for switching. The attribute "empty" of the tank is observable. Assign observers to all the tanks for the attribute "empty" and assign to all tanks the method switchTank. Between the closing and opening of the pipes, pass a small amount of time, so that Plant Simulation "can follow." The method switchTank might look like this:

```

(Attribut: string; oldValue: any)
is
do
  if ? = tank1 and ?.empty then
    p4.PipeOpened:=false;
    wait(0.001);
    p5.PipeOpened:=true;
  elseif ? = tank2 and ?.empty then
    p5.PipeOpened:=false;
    wait(0.001);
    p6.PipeOpened:=true;
  elseif ? = tank3 and ?.empty then
    p6.PipeOpened:=false;
    wait(0.001);
    p4.PipeOpened:=true;
  end;
end;

```

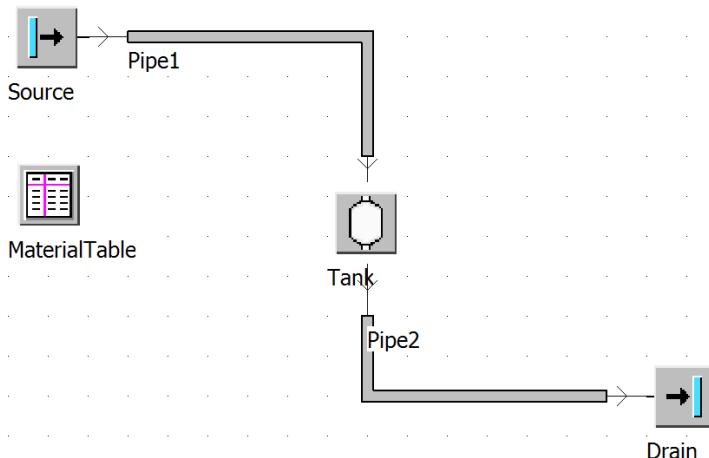
## 10.3 Simple Case Studies

### 10.3.1 Pumping Out

There are a number of applications in which the inflow rate of the tank is lower than the outflow rate. Thus, e.g. products are collected in a tank first until a certain level and then pumped off within a relatively short time. If you try to model such a situation, you receive an error message.

**Example: Fluids—Pumping out**

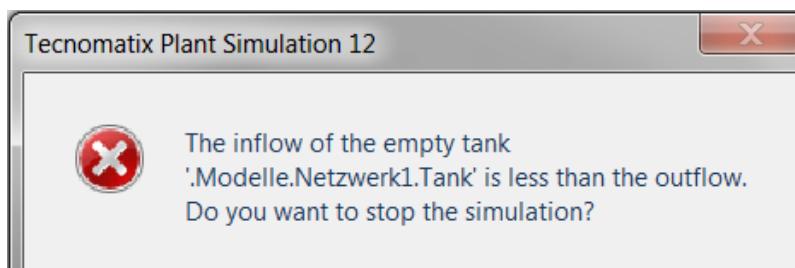
Create a simple frame like in Fig. 10.7.



**Fig. 10.7** Example frame

Enter the following settings:

Source: outflow rate: 0.25 l/sec; tank: 10,000 l volume, outflow rate: 10 l/sec. If you start the simulation, you will get an error message (Fig. 10.8).



**Fig. 10.8** Error message

If the tank has no connector to the next pipe while filling, it can be filled easily. To pump it, you would need to create a connector temporarily. Complete the frame as in Fig. 10.9.

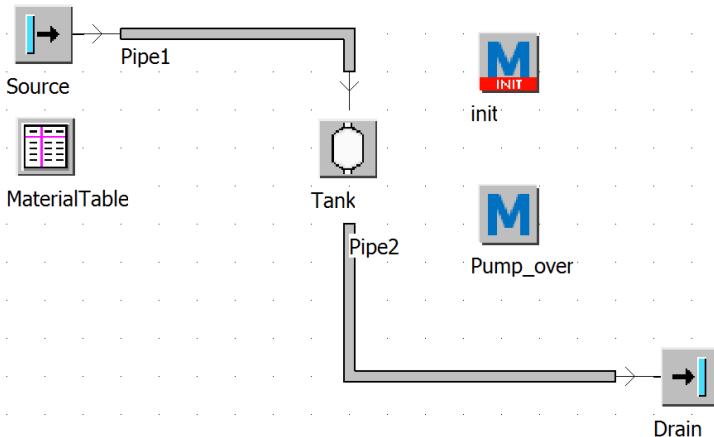


Fig. 10.9 Example frame

Add an observer for the attribute "full" to the tank and assign the method Pump\_over. This method creates a connector between the Tank and Pipe2, and deletes it when the tank is empty:

```
(Attribut: string; oldValue: any)
is
  conn:object;
  t:time;
do
  if oldValue=false then
    --calculate the time for pumping
    t:=?.volume/?.outflowrate;
    --create connector
    conn:=.MaterialFlow.Connector.connect(?,pipe2);
    wait(t);
    conn.deleteObject;
  end;
end;
```

In the method init, the connector between the Tank and Pipe2 must be deleted if it exists:

```
is
do
  if tank.numSucc > 0 then
    tank.succConnector(1).deleteObject;
  end;
end;
```

### 10.3.2 Distribute Fluids

With the help of dynamic connectors, you can model distribution tasks.

#### Example: Fluids—Distribution

In the following example, a cyclic loading with constant removal is to be simulated. Create a frame like in Fig. 10.10.

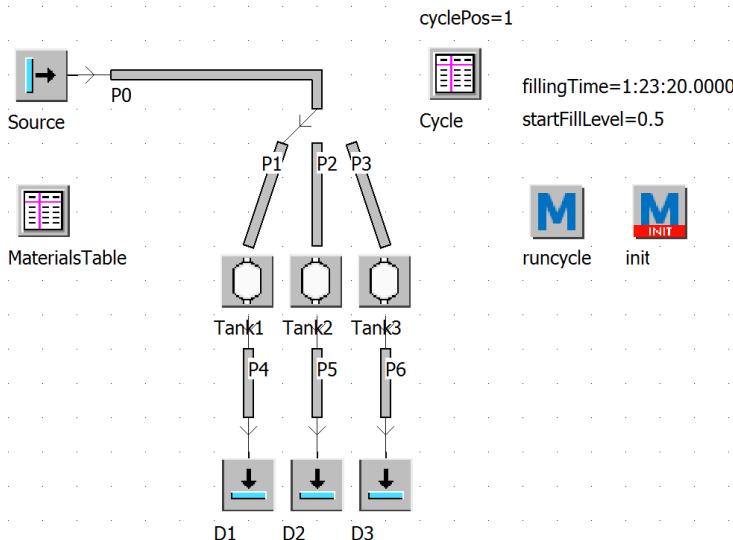


Fig. 10.10 Example frame

Prepare the following settings:

Source: outflow rate: 3 l/s; tanks: volume: 30,000 liters each, outflow rate: one l/s. The table Cycle contains the pipes P1 to P3 as object references. Format the table Cycle and enter the content as shown in Fig. 10.11.

object	
1	P1
2	P2
3	P3

Fig. 10.11 Table Cycle

In the init method, the tank should be filled to a certain level (fillingLevel). Thereafter, the first connection (p0 → p1) should be generated and the first fill cycle can be started. At the end, runCycle is called to start the periodic filling:

```

is
do
  --initial tank content
  tank1.setCurrentContent(
    tank1.volume*startFillLevel,
    "StandardMaterial", materialsTable);
  tank2.setCurrentContent(
    tank2.volume*startFillLevel,
    "StandardMaterial", materialsTable);
  tank3.setCurrentContent(
    tank2.volume*startFillLevel,
    "StandardMaterial", materialsTable);
  --delete connector of pipe p0
  p0.succConnector(1).deleteObject;
  --first connection
  .MaterialFluss.Kante.connect(p0,p1);
  wait(fillingTime);
  cyclePos:=1;
  runCycle;
end;

```

The method `runCycle` increases the cycle counter (`cyclePos`), deletes the current connector of `P0` and generates a new connector with the pipe at the current cycle position. After expiration of the filling time (`fillingTime`), the `runCycle` method is called again; thus, the next tank is filled. Method `runCycle`:

```

is
do
  cyclePos:=cyclePos+1;
  if cyclePos > cycle.yDim then
    cyclePos:=1;
  end;
  --delete connector of P0
  p0.succConnector(1).deleteObject;
  -- create connector to the next tank
  .MaterialFlow.Connector.connect(
    p0,cycle[1,cyclePos]);
  --switch to the next tank after filling time
  self.methCall(fillingTime);
end;

```

### 10.3.3 Fill the Tank with a Tanker

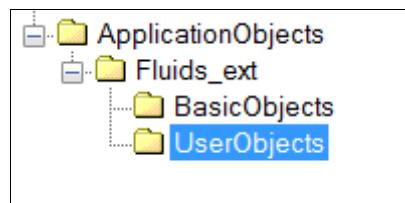
Tanks are often filled using tank trucks. To simulate this, you would need to connect the elements from the event-oriented simulation (track, transporter, buffer) with the continuous simulation (FluidSource, pipe, tank).

## Modeling Approach

For a tank truck, you can modify a transporter. You will need some information that helps to control a FluidSource. If you "pumped over" the contents of the tank truck, you create the required amount of liquid with the FluidSource using the pump power of the tanker. To use this function in different models, develop the filling station as a library element.

### Create a Library

First, place a folder named ApplicationObjects in the class library directly in the root. In the folder ApplicationObjects, create a folder named Fluids\_ext (which is the library). The library will contain all the basic elements that you need to create your special classes and the new elements. Create a folder structure as in Fig. 10.12.



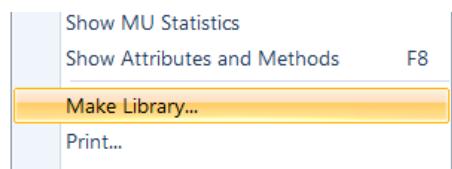
**Fig. 10.12** Folder structure

Duplicate a transporter in the BasicObjects folder. Rename the transporter in SiloTruck. As in a tank, you will need to extend the transporter with some attributes for handling the fluids (Fig. 10.13).

Name	Value	Type	C.	L.
currentFillAmount	0	real	*	
InflowRate	16	real	*	
OutFlowRate	33.33	real	*	
pipeOpened	false	boolean	*	
volume	20000	real	*	

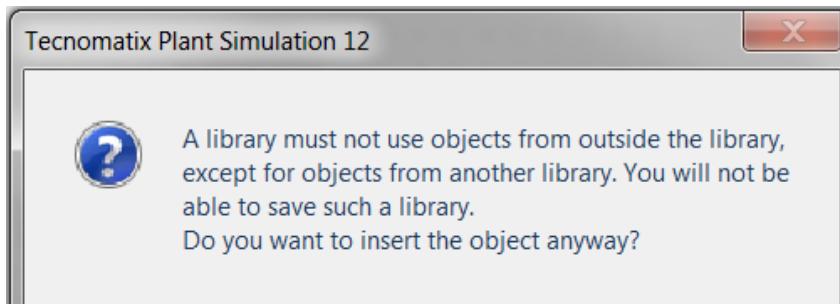
**Fig. 10.13** User-defined attributes of the SiloTruck

CurrentFillAmount stores the currently loaded amount, InflowRate and OutFlowRate store the flow rates, and PipeOpened indicates whether the fluid is currently in motion. Volume sets the volume of the tank truck. Since the unloading station of the tanker is composed of several blocks, you have to create a frame. Create a frame in the folder UserObjects. The name of the frame is SiloTruckUnloading. If you want to create a library item, you can use only items from the library. Faults in the structure are relatively difficult to correct later. Therefore, before you start to build the elements, convert the folder Fluids\_ext into a library (only professional license). To do this, select the Make Library option from the context menu of the folder (Fig. 10.14).



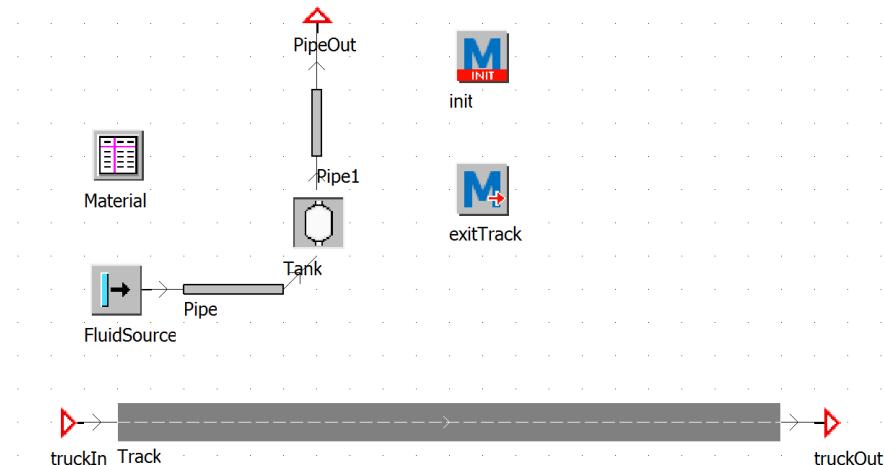
**Fig. 10.14** Make Library

If you now try to insert elements from outside the library in a library item, Plant Simulation shows an error message (Fig. 10.15).



**Fig. 10.15** Plant Simulation error

Create, as in Fig. 10.16, the frame SiloTruckUnloading. Duplicate all the elements that you need in the folder BasicElements (including the connector).



**Fig. 10.16** Frame SiloTruckUnloading

The init method closes the Pipe:

```
is
do
  pipe.pipeOpened:=false;
end;
```

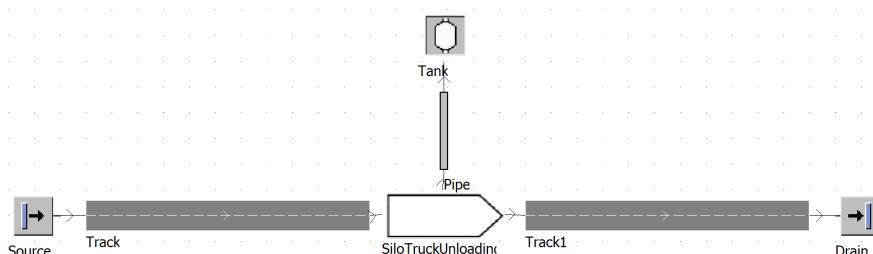
ExitTrack is the exit control of the Track. The unloading of the tank truck is modeled as follows: First, the outflow rates of the source and the tank are set according to the rate of the transporter. The time to pump is calculated from the volume of the transporter and the outflow rate. The pipe is opened and closed again after the unloading time. Thus, the amount of fluid from the tank truck is generated for the simulation. This yields the following program for the method exitTrack:

```
is
  t:time;
do
  --set flow rates
  fluidSource.outflowRate:=@.outFlowRate;
  tank.outflowRate:=@.outFlowRate;
  --calculates the unloading time
  t:=num_to_time(@.currentFillAmount/@.outFlowRate);
  --after unloading time close the pipe
  pipe.pipeOpened:=true;
  wait(t);
  pipe.pipeOpened:=false;
  @.currentFillAmount:=0;
  --source statAmount
  @.move;
end;
```

### Test Model

To perform a test, you would need a small model to examine the behavior of the tank truck.

The tanker should stop. The content of the tank truck should appear in a connected tank as inventory at the end of the pumping process. Create a frame as in Fig. 10.17. Use the class SiloTruckUnloading.



**Fig. 10.17** Example frame

Set `currentFillAmount` of the truck in the class library to 20,000 liters. Adjust the source so that exactly one `SiloTruck` is generated. Set the volume of the tank to 50,000 liters. Set `diesel` as the material in the `FluidSource` within the `SiloTruckUnloading` station and generate an appropriate entry in the `MaterialsTable`. If you have done everything right, then the tanker drives to the unloading station and fills the tank. At the end, the tank should have a stock of 20,000 liters (Fig. 10.18).

Current material:	Diesel		
Current fill level:	40.00%	Current inflow rate:	0 l/s
Current amount:	20000.000 l	Current outflow rate:	0 l/s

Fig. 10.18 Statistics of the tank

### 10.3.4 Unload a Tank Using a Tanker

Analogous to emptying the tank truck, it is not possible to load the fluid on the truck. Instead, you must destroy the required amount of fluid with a `FluidDrain` and transfer the information to the transporter. Create a frame (`SiloTruckLoading`) as in Fig. 10.19 in the class library.

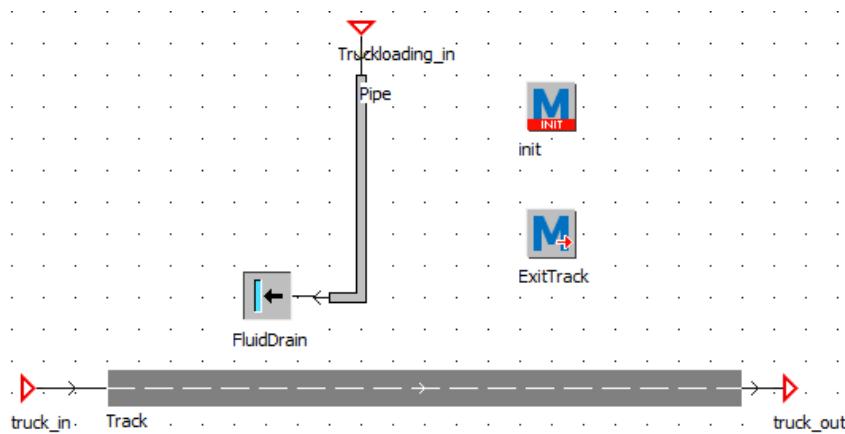


Fig. 10.19 Frame SiloTruckLoading

The method ExitTrack is the exit control (front) of the track. The init method closes the pipe at the beginning of the experiment:

```
is
do
  pipe.pipeOpened:=false;
end;
```

When the truck arrives, first adjust the outflow rate of the connected tank to the InflowRate of the truck. Then, calculate the time required for loading the truck. The pipe is then opened and fluid destroyed from the predecessor tank for this iteration. After the end of the filling process, the pipe is closed again and the current amount of the tank truck is updated. This results for the method ExitTrack in the following program:

```
is
  tank:object;
  t:real;
do
  --set outflowrate in the predecessor of pipe
  tank:=pipe.pred;
  tank.outflowrate:=@.inflowRate;
  --calculate time
  t:=@.volume/@.inflowrate;
  --start loading
  pipe.pipeOpened:=true;
  wait(t);
  pipe.pipeOpened:=false;
  @.currentFillAmount:=@.volume;
  --drive
  @.move;
end;
```

Set in the tank truck a volume of 20,000 liters and an InflowRate of 16. The attribute currentFillAmount has a start value of zero. Create with the SiloTruckLoading station a frame like in Fig. 10.20.

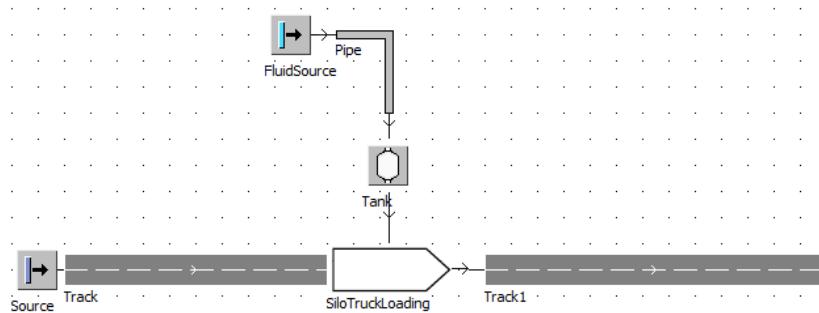


Fig. 10.20 Example frame

Prepare the following settings: The source generates after two hours exactly one tank truck. The FluidSource has an outflow rate of 5 l/s. The tank has a volume of 50,000 liters. Check the stock of the tank before and after the loading of the truck.

### 10.3.5 Separator

There are a number of processes in which the material flow branches out. In some cases, change the properties of the materials so that other materials leave the element. Examples include, e.g. sieves to sort materials by size, separators, etc. In the simulation, one material flow should split up into two flows.

#### Example: Fluids—Separator

In a separator, dust and debris is separated from grain. Here, the dust portion is five per cent. The separator shall be modeled as a library item.

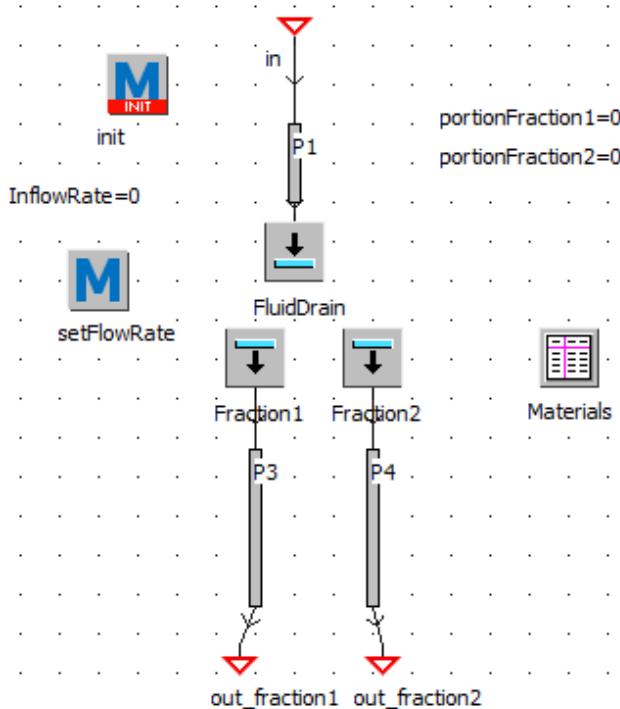


Fig. 10.21 Separator

### Modeling Approach

The simplest approach is to destroy the incoming material and recreate the fractions of the material. Thus, the separator contains a sink and two sources. In addition, you need to configure the proportions of the fractions and a MaterialsTable. Create as in Fig. 10.21 a frame in the class library.

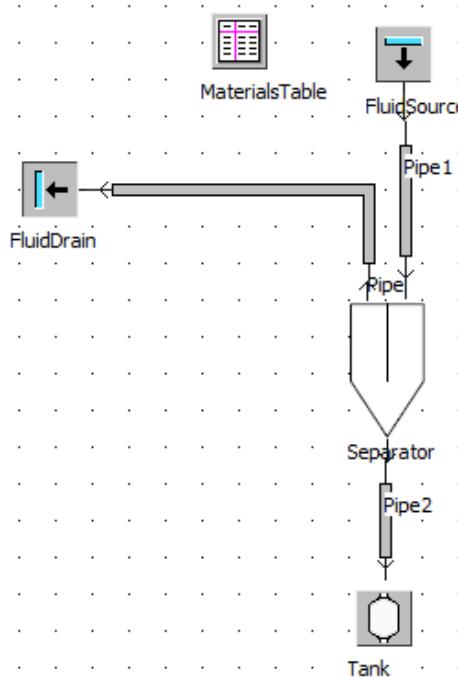
The method `setFlowRate` is the observer control for the attribute `currentFlowRate` of the pipe `P1` (Tools—Edit Observer). By monitoring the attribute `currentFlowRate`, you can react to incoming material. If material arrives, set the outflow rates of the sources `Fraction1` and `Fraction2` depending on the inflow rate of `P1` and the proportions of the fractions. Then, open the pipes `P3` and `P4`. This yields the following programming for the `setFlowRate` method:

```
(attribute: string; oldValue: any)
is
do
  if ?.currentFlowRate > 0 then
    inFlowRate:=?.currentFlowRate;
    fraction1.outFlowRate:-
      inflowRate*portionFraction1/100;
    fraction2.outFlowRate:-
      inflowRate*portionFraction2/100;
    p3.pipeOpened:=true;
    p4.pipeOpened:=true;
  else
    p3.pipeOpened:=false;
    p4.pipeOpened:=false;
  end;
end;
```

In the `init` method, you must close the pipes `P3` and `P4`.

```
is
do
  p3.pipeOpened:=false;
  p4.pipeOpened:=false;
end;
```

To test it, create a frame such as in Fig. 10.22.



**Fig. 10.22** Example frame

Prepare the following settings:

MaterialsTable—insert material "Grain" (1 g/cc), and assign the MaterialsTable and the material Grain to the FluidSource (outflow rate 1 l/s). The tank has a capacity of 50,000 liters. In the separator, ensure the following settings: portionFraction1: 5, portionFraction2: 95, Fraction1: Material Dust, Fraction2: Grain, materials like in Fig. 10.23.

string	Material	Density [g/cm <sup>3</sup> ]	Color	Product Amount	Unit
1	Dust	0.800	49407	1.000	kg
2	Grain	1.200	16729600	1.000	kg

**Fig. 10.23** MaterialsTable Separator

### 10.3.6 Status Change

In some processes, transitions are required between the event-oriented and process-oriented simulations. This may include the exchange between cargo and bulk material (e.g. by breaker) or intermediate products, which are handled as a piece and then return into the process.

### Example: SolidInterface

In a cookie factory, dough prepared in the first step must mature for a while. The dough is shaped by a portioning system into biscuits and baked. In the first step of the portioning, the device SolidInterface is developed. For the model, you need two attributes in the Entity (volume: real; Material: string).

### Modeling Approach

For a connection between Entity and fluid, the Entity must be destroyed and a specific amount of fluid created. This works most easily with the help of a drain and a LiquidSource. The drain destroys the MU. The FluidSource generates fluids with the volume of the MU and passes it to a tank. Create in the class library a frame (SolidInterface) according to Fig. 10.24.

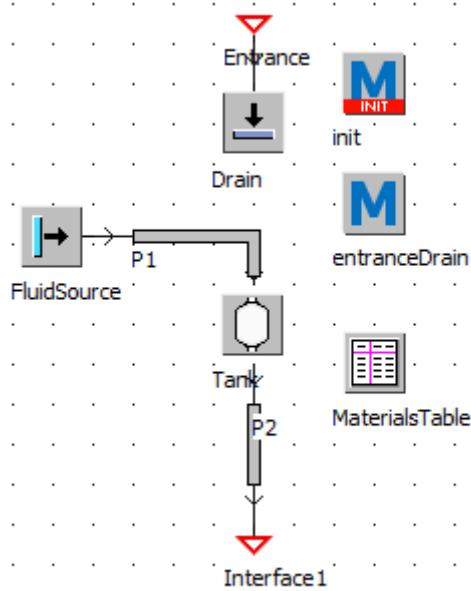


Fig. 10.24 Frame SolidInterface

The method entranceDrain is the entrance control (before action) of the drain. The entrance control first closes the entrance of the drain. Then, it waits until the tank is empty. The duration is then computed, which is necessary to produce the volume of the MU via the FluidSource. For this duration, P1 is opened. At the end, the entrance of the drain is released and the drain destroys the MU. The method entranceDrain might look like this:

```
is
  t:time;
```

```

do
  --block the entrance of the drain until
  --the tank is empty
  drain.entranceLocked:=true;
  waituntil tank.empty prio 1;
  --transfer the volume of the MU
  t:=@.volume/FluidSource.outflowRate;
  FluidSource.material:=@.material;
  p1.pipeOpened:=true;
  wait(t);
  p1.pipeOpened:=false;
  drain.entranceLocked:=false;
end;

```

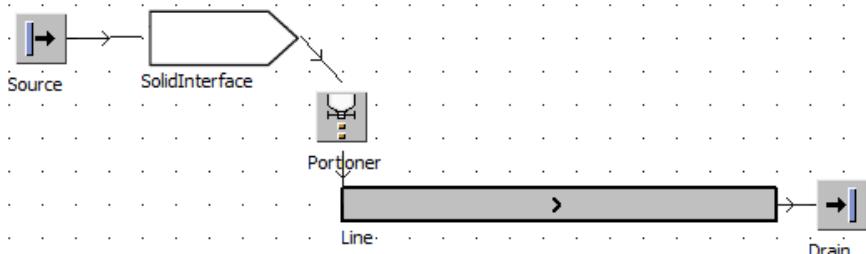
The init method must unlock the entrance of the drain and close P1:

```

is
do
  drain.entranceLocked:=false;
  p1.pipeopened:=false;
end;

```

Create, using the SolidInterface to test, a frame similar to Fig. 10.25:



**Fig. 10.25** Example frame

Ensure the following settings: The source produces Entities every 25 minutes. The Entity has a volume of 150 liters and Dough as material. In the SolidInterface, adjust the FluidSource to an outflow rate of 100 l/s. The tank holds 200 liters and has an outflow rate of 0.1 l/s. Assign the MaterialsTable to the FluidSource. Insert Dough as material in the MaterialsTable. The Portioner requires 0.01 liters of material per MU. The line has a speed of 0.06 m/s.

# Chapter 11

## 2D and 3D Visualization

Good animation can increase significantly the acceptance of a simulation. Plant Simulation supports both 2D and 3D visualization. The explanations in this chapter refer to the versions 12 and later.

### 11.1 2D Visualization

It is helpful to use a realistic layout for the simulation model. The layout is divided into several levels:

1. The background of the frame
2. The icons of the static objects
3. The icons of the mobile objects and their movement through the frame

All objects of a class share a set of icons. Therefore, you have to duplicate enough classes in the class library for representing a more complex model. For a large quantity of objects, it is recommended to organize them in folders or libraries.

#### 11.1.1 *The Icon Editor*

Every frame and almost every basic object has a set of icons.

##### **Example: Icon editor**

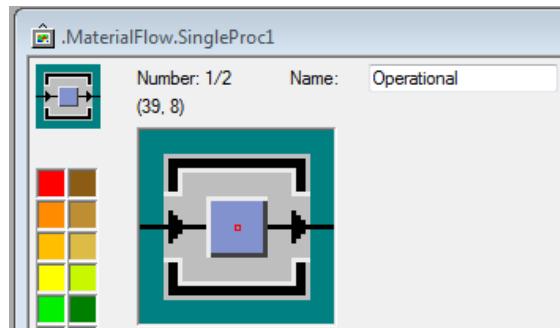
Duplicate a SingleProc in the class library. Open the context menu by clicking the object with the right mouse button and select Edit Icons. It opens the icon editor (Fig. 11.1).

Most objects have two icons:

- No. 0 is the icon that Plant Simulation displays in the toolbar.
- No. 1 is the icon that Plant Simulation displays when you insert the object in a frame.

You can always extend the pool of icons. Select New in the tab Edit. You can replace the existing icon in the following ways:

- You could draw a new icon. To ensure that the icon is displayed without limitation in the tools (No. 0) it may have a maximum size of 40 x 40 pixels. All other icons are limited to a size of 9999 x 9999 pixels.
- You can import existing graphics (file or clipboard).



**Fig. 11.1** Icon editor

## 11.1.2 *Inserting Images*

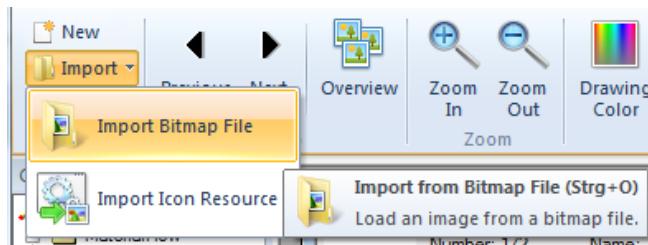
### 11.1.2.1 Insert Images from the Clipboard

You can insert pictures through copy and paste. Keep in mind that you cannot control the size of the picture after inserting it into Plant Simulation. Copy the image to the clipboard. In Plant Simulation select **Ctrl +V**. You can also insert images into the icon editor using drag and drop. Drag the icon file from the Windows Explorer window into the icon editor. Plant Simulation changes the size of the icon automatically. If necessary, the image can be modified in the icon editor.

### 11.1.2.2 Inserting Images from a File

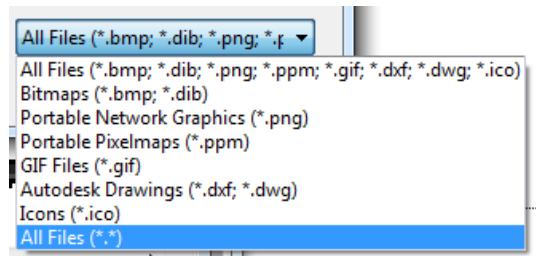
You can also create the object icons from existing files.

Select **File—Import—Import Bitmap File** in the tab **Edit** (Fig. 11.2).



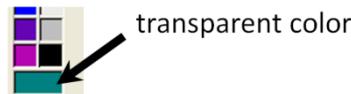
**Fig. 11.2** File import

Plant simulation supports importing a wide range of file types (Fig. 11.3).



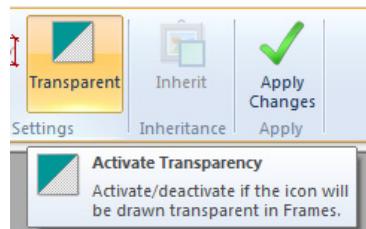
**Fig. 11.3** Supported file types

You can even convert CAD drawings into pixel images. When you insert an image from the clipboard or from a file, the transparency information in GIF and PNG files is lost. Plant Simulation provides a transparent color (dark green) located at the bottom of the color palette (Fig. 11.4).



**Fig. 11.4** Transparent color

Fill all pixels that should be transparent in the icon with this color. Activate the transparency of these pixels with the command Transparent in the tab Edit (Fig. 11.5).

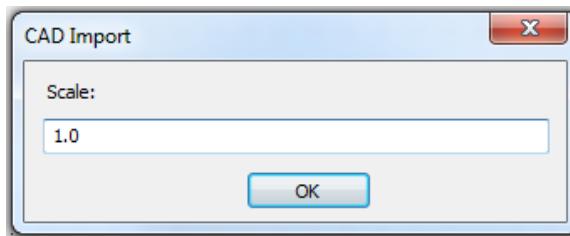


**Fig. 11.5** Transparent

### 11.1.2.3 Changing the Background of the Frame

For frames, you can define an icon with the name "background." This icon is shown as the background of the frame. The easiest way is to drag an image file from the Windows Explorer into a frame. Plant Simulation then creates the icon Background and inserts the content of the image file as the background icon. If you want to use an AutoCAD file as background, you must first set the scaling factor of the frame (General—Scaling Factor). If you drag the AutoCAD file into the frame, Plant Simulation asks you for a scaling factor for the CAD file

(Fig. 11.6). The best results are achieved if the AutoCAD File is created in mm, you use 1:1 for the import and set the scaling factor in Plant Simulation.



**Fig. 11.6** Scaling factor CAD import

### 11.1.3 Animation Structures and Reference Points

When you create your own icons, you have to determine where the MU is displayed on the icon and how the MU (e.g. on a track) will move on the icon. You ensure these settings:

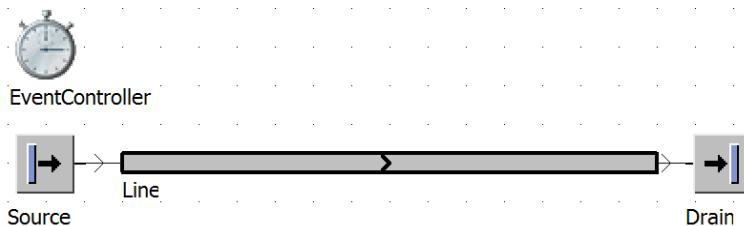
- With a reference point on the MU
- With an animation structure on the material flow objects

#### 11.1.3.1 Set Reference Points

The reference point determines where Plant Simulation displays the icon of the part when you insert the object into the frame or during the animation on another object. If it is an MU, it will be positioned so that its reference point lies on an animation point or an animation line. If the object is a basic object or a frame, it will be aligned to the grid in a frame window. The reference point is positioned on a grid point. The reference point is displayed as a red pixel in the icon editor. You can move the reference point by clicking the Set Reference Point button and by clicking a new pixel.

#### Example: Reference Points and Animation Structures

Create a frame like in Fig. 11.7.



**Fig. 11.7** Example frame

Duplicate the entity in the class library (right mouse button—Duplicate). Name the new class "Part". Disable in the dialog box of the part the option Graphics—Vector graphics active. The source creates parts in intervals of 10 seconds. The line has a speed of 0.1 m/sec. Set the width of the line to nine pixels.

The icon of the part should have a size of 7 x 7 pixels and the color should be green. Open the icon editor (right click—Edit icons). First, change the icon size. Select the menu item: Edit—Size. Enter a height and a width of seven and click OK (Fig. 11.8).

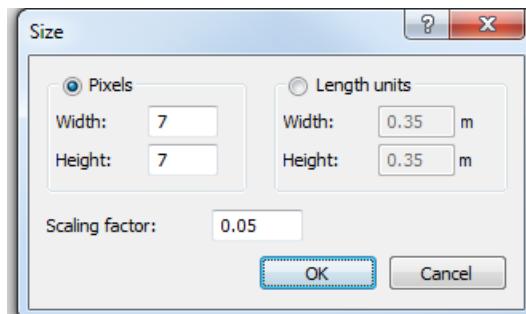


Fig. 11.8 Icon size

Click on a green color. Then use a filled rectangle to create a green icon. Apply your changes. Test the model. The MUs now are aligned and not centered on the conveyor line (the orientation is still from the previous symbol; see Fig. 11.9).

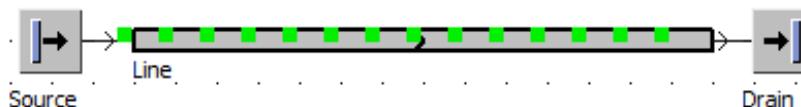


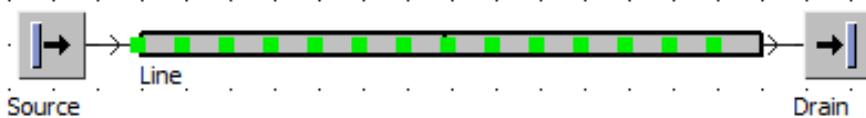
Fig. 11.9 Wrong alignment

You have to reset the reference point (in the middle of the icon at the position on 4 x 4 pixels). Click Edit—Reference Point. Then select the pixel at the position 4,4 (Fig. 11.10). The reference point is framed in red.



Fig. 11.10 Reference point

Now, the alignment of the MUs is centered on the line (Fig. 11.11).



**Fig. 11.11** Alignment centered

Note: If you change the size of an MU, you have to do this for the entire set of icons.

The entity has two icons that you have to change:

- operational (number 0)
- waiting (number 1)

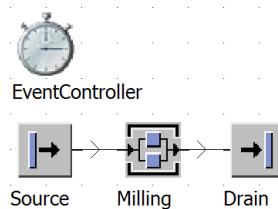
The fastest way is to copy the current icon (Ctrl +C), switch to the next icon and then paste it again (Ctrl +V). You have to set the reference point in the second icon (before Plant Simulation Version 8).

### 11.1.3.2 Animation Structures

In the icon editor's Animation tab, you can set animation points or animation lines.

#### Example: Animation structures

Duplicate a ParallelProc in the class library. Rename the object as "press". Set the capacity to two. Create with the press a simple frame as per Fig. 11.12.



**Fig. 11.12** Example frame

Open the icon editor for Press (starting from the class library). Select General—Clipart—Training—MillingPic. Apply your changes and run the simulation. The MUs are not displayed at the correct positions (Fig. 11.13).

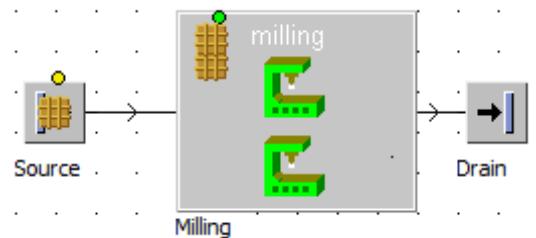


Fig. 11.13 Wrong MU position

Open the icon editor for milling in the class library. Click on the folder Animation. The icon will be shown grayed out. The built-in ParallelProc already has four animation points. If you change the size of the icon, then you have to manually change the animation points. Delete all animation points (Delete all). Insert two animation points like in Fig. 11.14 (click first on Point, then on the right position in the icon).

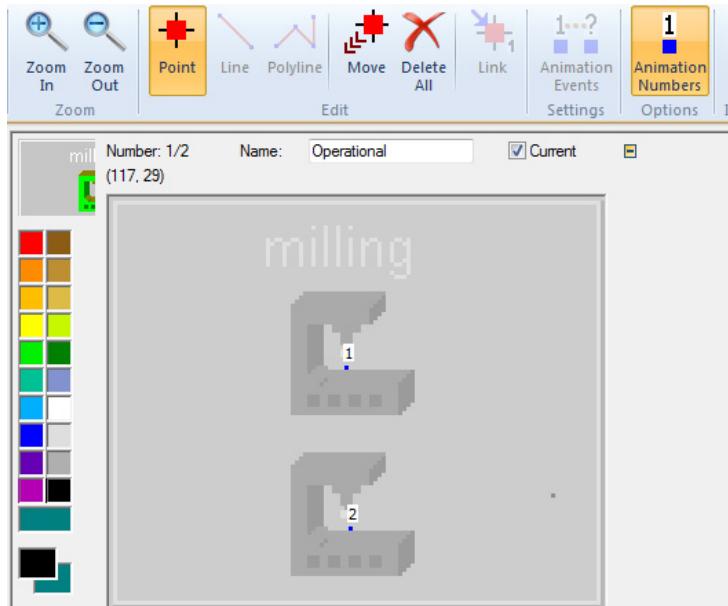
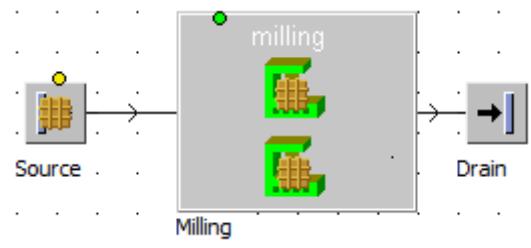


Fig. 11.14 Animation mode

The MUs are now animated exactly at these positions on the block. Apply your changes and test your model (Fig. 11.15).



**Fig. 11.15** MU animation

PlaceBuffer, Line, Track and Sorter use animation lines and animation polylines. If you also want to watch animation, you have to specify how often the MU will be displayed on this line.

Example: Select the PlaceBuffer and open the icon editor. Switch to animation mode. You see a line on the icon. If, for example, more than one animated part is located on the block, you have to specify a number of animation events greater than one. Click on Animation—Animation events. Enter in the following menu a number less than or equal to 250. The default is one, which means that the MU is animated at the beginning and at the end of the block, not in between.

#### 11.1.4 Animating Frames

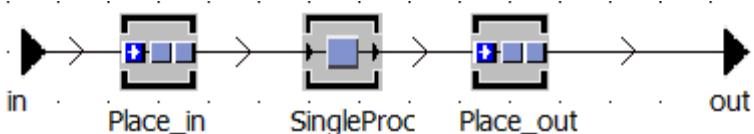
Often, it is necessary to split simulations that are complex. The simulation takes place in various parts or segments, which are connected with each other through connectors. At the top level is an overview of all the objects (frames) of the simulation. For a better presentation of the sequences, you can animate the flow of mobile units on the frame.

##### Example: Machine with a Ring Loader, Animation on a Frame

You want to simulate a machine. The machine is equipped with a ring loader, which offers 15 places each for unfinished and finished parts. The ring loader is "accumulating". Insert a frame (e.g. Machine). To simulate this machine, you need (for example) three objects:

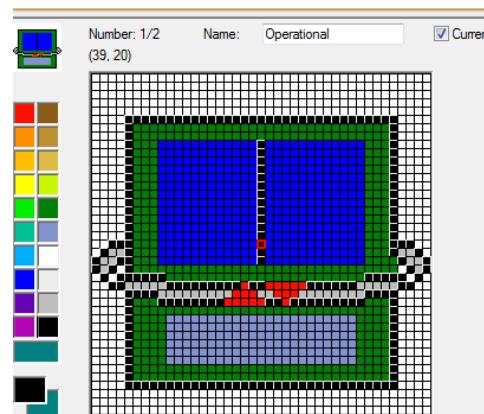
- EntranceBuffer
- SingleProc (processing time: one minute)
- ExitBuffer

Interface objects are necessary for connections with other objects. The frame for the machine might look like Fig. 11.16.



**Fig. 11.16** Sub-frame machine

Change the frame icon (so that it indicates the ring loader, Fig. 11.17).

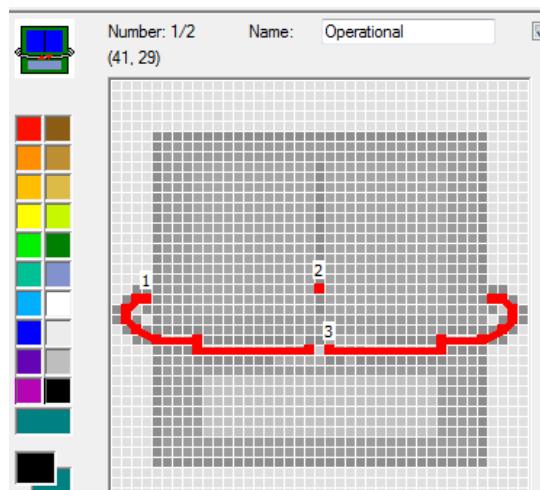


**Fig. 11.17** Icon ringloader

Change to animation mode: Draw three structures on the icon. One each for:

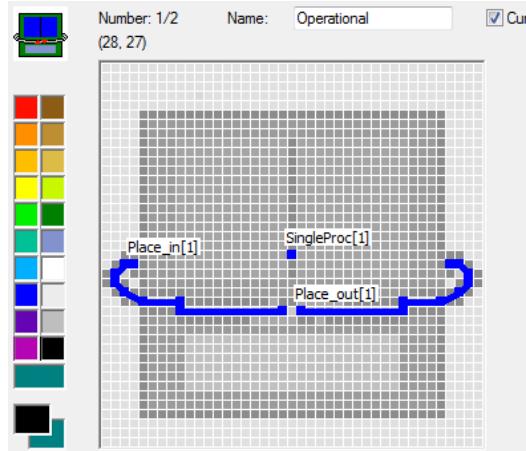
- EntranceBuffer
- SingleProc (processing time: 1 minute)
- ExitBuffer

You can do this with polylines. Click on the icon Polyline. Then click on separate points; Plant Simulation connects these points with a polyline. A right click finishes the polyline. Plant Simulation numbers your animation structures. At the end, it should look like Fig. 11.18.



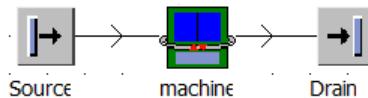
**Fig. 11.18** Animation structures

Now, the animation structures on the icon of the frame must be connected with the animation structures of the objects within the frame. First, click on the icon Link. Then, click on the structure for the input buffer (number 1). It opens a window in which you have to select the respective object and its animation structure. The names of the objects will be listed on the animation structure (Fig. 11.19).



**Fig. 4.19** Linked animation structures

For testing, build a frame like in Fig. 11.20.



**Fig. 11.20** Example frame

The MUs are now animated on the object.

### 11.1.5 Dynamic Creation of 2D Animation Structures

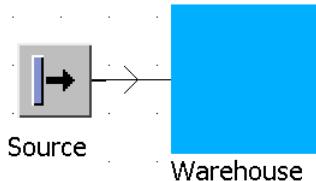
If you need to generate a large number of animation points, then the uniform distribution of the animation points is a big challenge. SimTalk provides methods and attributes for generating 2D animation structures automatically. The attributes and methods from Table 11.1 help you in the creation of animation points and the manipulation of symbols:

**Table 1** 2D animation and SimTalk

Attribute/Method	Description
<path>.currIconNo	Sets the current icon (integer)
<path>.currIcon	Sets the current icon (string name)
<path>.getIconSize( byRef int height, byref int width)	Gets the size of the current symbol, you must pass one variable for width and one for height
<path>.setIconSize(int width, int height)	Sets the size of the current icon
<path>.getPixel(int x, int y)	Returns the color of the pixel at the position x,y; the return value has the type integer
<path>.setPixel(int x, int y, int color)	Sets the color of the pixel at the position x,y
<path>.delAniPoints	Deletes all animation points in the current icon
<path>.setAniPoint (int x, int y)	Sets a new animation point with a distance x,y from the upper left corner
<path>.zoomX, <path>.zoomY	Returns the zoom factor of the icon

**Example: Dynamic Creation of Animation Points**

In the following example, the animation of the store block should be simplified. First, duplicate the store block. The following programming is done in the duplicate of the store block (warehouse) in the class library. When scaling the block in the frame, the animation structure should be adapted automatically according to the set capacity of the store block. Create with the duplicate a (very simple) frame like Fig. 11.21.

**Fig. 11.21** Example frame

Create user-defined attributes in the warehouse block in the class library according to Fig. 11.22.

Name	Value	Type	C.	I.
createAniPoints		method	*	
imageHeight	0	integer	*	
imageWidth	0	integer	*	
onCapacityChange		method	*	
onZoom		method	*	

**Fig. 11.22** User-defined attributes

## Enlarge Icon at Zoom

In the first step, the icon should be enlarged if you zoom in on it in the frame. This is necessary in order to make enough room for the animation points.

You can zoom in on icons in the frame by pressing Ctrl + Shift and dragging with the mouse. However, the underlying icon will not increase. Zooming changes the attributes zoomX and zoomY. To enlarge the icon according to the zooming you need to take the following steps:

1. A method is called via an observer for the attributes zoomX and zoomY.
2. The icon size is read.
3. The icon height and icon width is multiplied by the associated zoom factor.
4. The icon size is allocated to the icon. The zoom factor is reset to one.
5. The color of the first pixel is read.
6. All pixels of the icon are colored with the color of the first pixel.
7. At the end, the user-defined method createAniPoints is called.

If you increase the size of an icon with SimTalk, the enlarged area is initially transparent. Therefore, a definition of the color for this area is necessary. To simplify, you can color the entire icon with one color.

First, add two observers for the warehouse block (select Tools—Select observers—Add), attributes zoomX and zoomY, and the method to be called as self.onZoom. The method onZoom should have following content:

```
(attribute: string; oldValue: any)
is
  h:integer;
  w:integer;
  color:integer;
  x,y:integer;
do
  if self.~.zoomX /=1 or self.~.zoomY /= 1 then
    -- read icon size
    self.~.getIconSize(w,h);
    -- multiply height and width with the zoom
    self.~.imageHeight:=self.~.zoomY*h;
    self.~.imageWidth:=self.~.zoomX*w;
    --change icon width
    self.~.setIconSize(self.~.imageWidth,
                      self.~.imageHeight);
    -- reset zoom
    self.~.zoomY:=1;
    self.~.zoomX:=1;
    --read color of the first pixel
    color:=self.~.getPixel(1,1);
    --color all pixels
    for x:=1 to self.~.imageWidth loop
      for y:=1 to self.~.imageHeight loop
```

```

        self.~.setPixel(x,y,color);
    next;
next;
--set new animation points
self.~.createAniPoints;
end;
end;

```

In the method `createAniPoints`, you must first delete all animation points of the current icon. To create the animation points you could first calculate the distance between the animation points and then create individual animation points. Thus, the user-defined method `createAniPoints` might appear as follows:

```

is
  x,y:integer;
  distanceY:integer;
  distanceX:integer;
do
  --delete all animation points
  self.~.delAniPoints;
  --calculate the distances between the points
  distanceX:=self.~.imageWidth/(self.~.xDim+1);
  if distanceX=0 then
    distanceX:=1;
  end;
  distanceY:=self.~.imageHeight/(self.~.yDim+1);
  if distanceY=0 then
    distanceY:=1;
  end;
  --create new animation points
  --row by row --> from left to right
  for y:=1 to self.~.yDim loop
    for x:=1 to self.~.xDim loop
      self.~.setAniPoint(x*distanceX,y*distanceY);
    next;
  next;
end;

```

In order to react to changing the `xDim` and `yDim` values, you need observers for these two attributes. Assign the method `self.onCapacityChange`. In this method, you only need to call the method `createAniPoints`. Method `onCapacityChange`:

```

(attribute: string; oldValue: any)
is
do
  self.~.createAniPoints;
end;

```

### 11.1.6 Simple 2D Icon Animations

With relatively little effort, you can create 2D icon animations (flipbook effect). To do this, the current icon is changed (attribute currIconNo) in a given time interval. If you are preparing an animation as single frames, then a "flipbook effect" results. Hence, you can, for example, animate the state of stations with "moving pictures."

#### Example: 2D Icon Animation

In the following example, animate an injection molding machine. The tool closes at the start of the processing. At the end of the processing, the tool opens and the part is ejected. Duplicate the SingleProc and insert into the duplicated SingleProc a set of icons as in Fig. 11.23.

Symbol-Nr	1	2	3	4	5	6	7	8	9
Symbol									

Fig. 11.23 Icons for icon animation

Delete from the symbols 1–5 the animation points, so that the part will appear in the animation only at icon 6. To program an animation, you need some values:

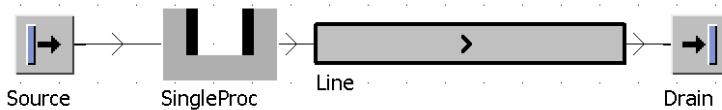
- Start symbol number (animationStartPic: integer)
- Icon-end number of the animation sequence (animationEndPic: integer)
- Frame refresh rate given as frames per second (framesPerSec: integer)
- A Boolean value indicating whether the animation will be repeated or not (animationLoop: boolean)

Set in the duplicated SingleProc in the class library the user-defined attributes and methods as in Fig. 11.24.

Name	Value	Type	C.	I
animate		method	*	
animationEndPic	9	integer	*	
animationLoop	false	boolean	*	
animationStartPic	1	integer	*	
framesPerSec	-1	integer	*	
init		method	*	

Fig. 11.24 User-defined attributes

Create a frame according to Fig. 11.25.



**Fig. 11.25** Example frame

Enter the method `animate` as the entrance control of the `SingleProc`. In the method `animate`, two cases are to be programmed:

- The animation runs exactly once.
- The animation is played throughout the time that the `SingleProc` is working.

When `-1` is set at `framesPerSec`, the animation duration should correspond with the processing time of the `SingleProc`. This results in the following programming:

```
is
do
  if self.~.framesPerSec = -1 then
    wait(self.~.procTime/(self.~.animationEndPic-
      self.~.animationStartPic+1));
  else
    wait(60/self.~.framesPerSec);
  end;
  if self.~.currIconNo < self.~.animationEndPic then
    self.~.currIconNo:=self.~.currIconNo+1;
    self.methcall(0);
  else
    self.~.currIconNo:=1;
    if self.~.animationLoop and
      self.~.occupied then
      self.methcall(0);
    end;
  end;
end;
```

The internal `init` method must set the icon back to the start of the animation:

```
is
do
  self.~.currIconNo:=self.~.animationStartPic;
end;
```

Similarly, you can also create animations on the base of the attribute `iconAngle`.

## 11.2 Plant Simulation 3D

With Plant Simulation, you can start from 2D models relatively easily to create 3D simulations. You can also model directly in 3D. A major problem with this, however, is the acquisition of 3D layouts for the system elements.

### 11.2.1 Introduction to Plant Simulation 3D

Plant Simulation supports modeling and simulating models in virtual space. The 2D model is assigned to the 3D model, which is controlled by the 2D model (corresponding models). All changes in the 2D/3D model have an impact on the corresponding model in the other part of the program.

#### Example: Basic 3D

You are to simulate two machines that are connected with a conveyor belt. The conveyor belt is divided into three segments of one meter each. The conveyor speed is 0.5 m/min, the processing time of the machines is one minute, and the source generates one part every minute. Create a frame like in Fig. 11.26.

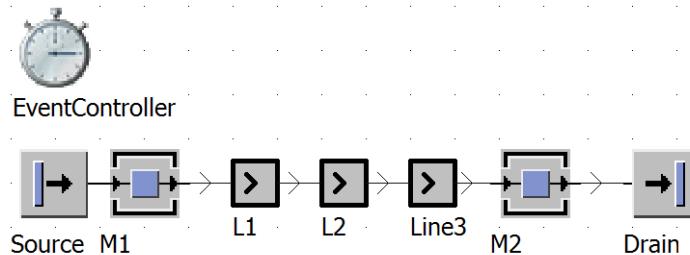


Fig. 11.26 Example frame

Click Home—Open 2D/3D (Fig. 11.27).



Fig. 11.27 Home—Open 2D/3D

You now have two corresponding models. The preliminary result is a 3D model with standard symbols (Fig. 11.28).

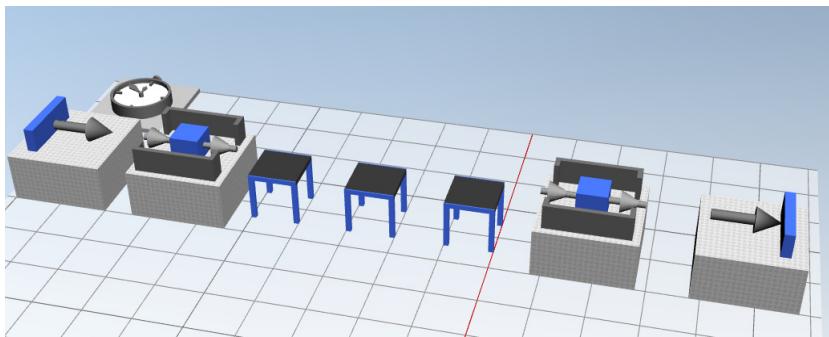


Fig. 11.28 3D—Model

### 11.2.2 Navigation in the 3D Scene

Corresponding to the size of the corresponding 2D simulation, Plant Simulation generates a base plate in the 3D model. As a 2D reference level ( $Z = 0$ ), the base plate facilitates orientation and navigation within 3D space. The base plate has a grid on which you can align your objects.

To navigate within the 3D network, you need the mouse (Table 11.2).

Table 11.2 Navigation 3D

Mouse function	Navigation
Mouse wheel	zoom
Right mouse button pressed down + move mouse	Moves the scene (pan)
Right + middle mouse button (wheel) pressed down + move mouse	Rotate scene

If you are lost, and your frame is no longer shown on the screen, use the command View—View All (Fig. 11.29).

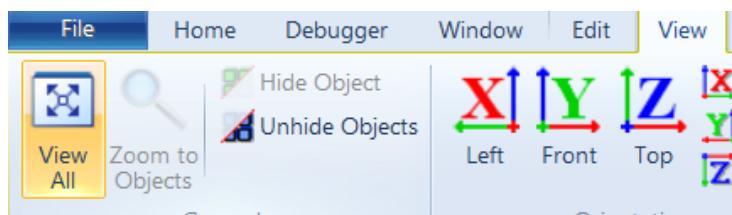


Fig. 11.29 View All

### 11.2.3 Formatting 3D Objects

Plant Simulation initially loads the standard 3D icons. You can load your own graphics and equip them with paths for animation.

### 11.2.3.1 Load 3D Graphics

Plant Simulation shows the elements initially with standard layouts that you can change at any time. You must select the objects in the frame (click). The object will be highlighted in green. You can open the 3D object by clicking "Open in new 3D window" in the context menu (right click). Via the Plant Simulation menu Edit—Import Graphics, you can insert a 3D graphic in the opened object. Plant Simulation supports the JT format from Version 9. You can also use VRML (.wrl). Plant Simulation converts those files into the JT format. You can return to the 3D frame via the context menu: Open Location or the key combination Shift + Enter.

#### Example: Basic 3D (continuation)

Select a SingleProc in the 3D viewer. Open the 3D object (context menu—"Open in new 3D window"). Select the graphic of the SingleProc and delete it (Delete key). Select in the main menu: Edit—Import Graphics (Fig. 11.30).

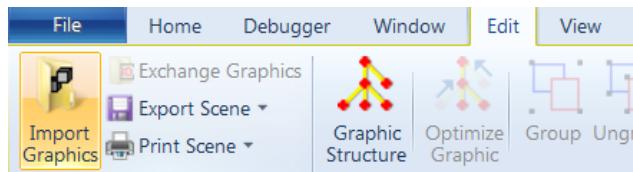


Fig. 11.30 Import Graphics

Plant Simulation automatically displays the files from the folder 3D/jt-graphics. Plant Simulation provides some graphics with the default installation. Select the file Workbench.jt and click Open. The 3D graphic is inserted into the opened object. Add a second graphic to the object (operatorStandingMale.jt, Fig. 11.31).

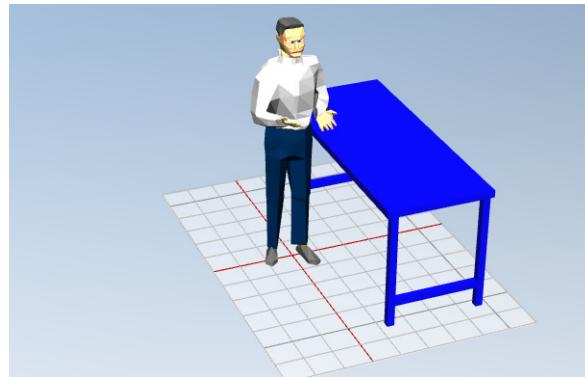
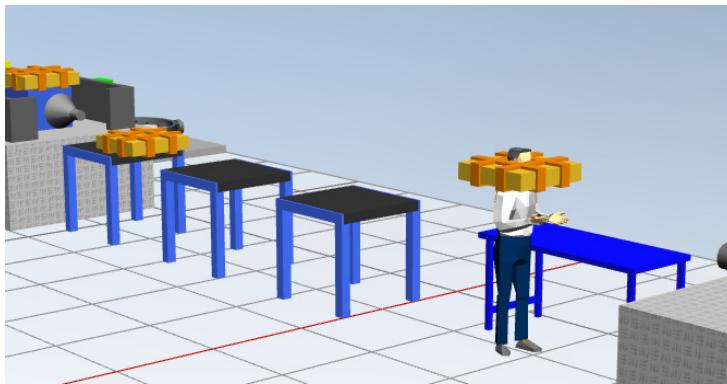


Fig. 11.31 Operator standing and workbench

You can move the selected graphics in the x and y directions using the arrow keys. Move the graphics as in Fig. 11.31 so that the operator is standing on the coordinate cross.

Change to the frame and start the simulation. The SingleProc is now displayed with the new graphics. But the animation of the MUs on the SingleProc is incorrect (Fig. 11.32).



**Fig. 11.32** Workbench and operator

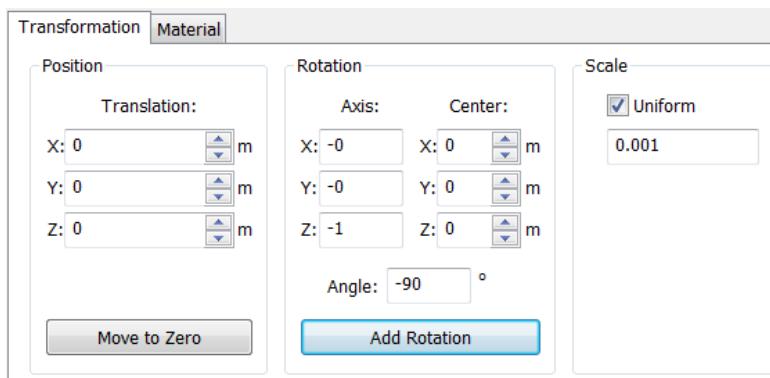
The MU is animated at the position of the original 3D graphic. The operator should be turned over to the table and keep the part in his hands. This needs to be corrected in the next step.

### 11.3.2.2 Transformations

Often, imported 3D geometries have a different origin than the graphics in 2D, on which the alignment is done in the 3D viewer. In such cases, the origin of the 3D graphics must be changed. Similar is the case if the orientation is wrong. In this case, the object must be rotated based on a point in space. These adjustments are made via the context menu of the objects command "Edit 3D Properties". You will also find an icon in the toolbar Home. In principle, you can move, rotate and scale 3D objects.

#### Example: Basic 3D (continuation)

Open M2 in a new 3D window. Highlight the operator and select from the context menu: Edit 3D Properties.



**Fig. 11.33** 3D properties

You can easily rotate and scale in this dialog. The operator has to be rotated by  $-90^\circ$  (counterclockwise  $90^\circ$ ) around the z-axis. Prepare the settings in Rotation from Fig. 11.33. As a result, the operator should now be in the right position.

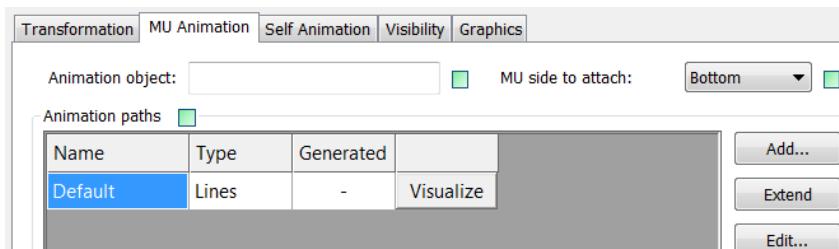
As in 2D, 3D objects have their own animation structures. If you change the 3D geometry, then you usually need to adjust the animation structures (paths).

### 11.3.2.3 Animation Paths 3D

You create and edit paths using the 3D properties of the object (in the current example, M2). Call the context menu of the object (in the background of the object, without marked graphic)—Edit 3D properties.

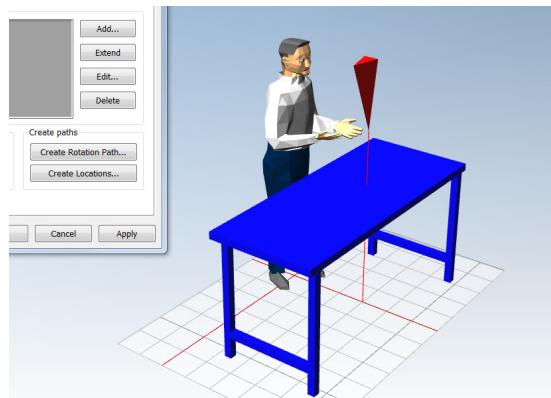
#### Example: Basic 3D (continuation)

Open the 3D properties of M2 and click on the tab MU Animation (Fig. 11.34).



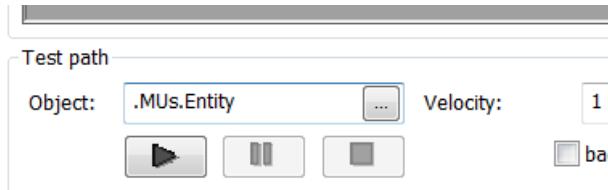
**Fig. 11.34** MU Animation

Click the Visualize button. The path (in the example, the animation point in red) is then displayed for the selected object. The path icon (red "triangle") must now be moved so that it is above the table (Fig. 11.35). The path symbol can be selected like other 3D objects by clicking on it (highlighted in green). Use the arrow keys to change the position of the path symbol. Position changes in the z-direction can be reached by pressing ALT + up and down arrow keys.



**Fig. 11.35 MU Animation**

Via Test path, you can try out your animation structure. Select in the field Test path—Object.MUs.Entity and press the button with the black triangle (Fig. 11.36).

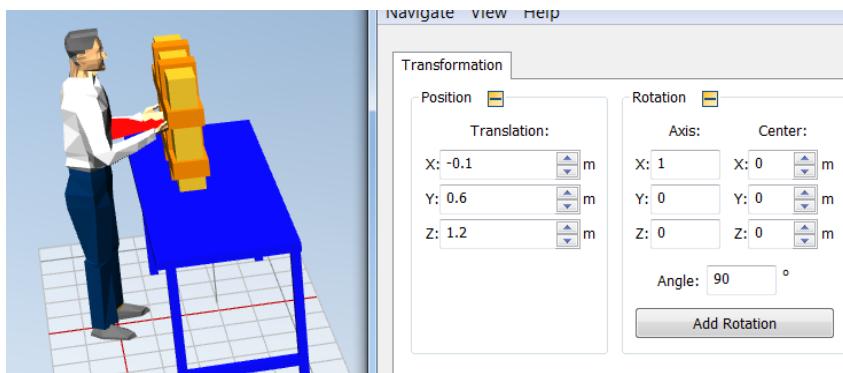


**Fig. 11.36 Test path**

The MU appears. Even now, you can display the path symbol. Select and move the path icon for precise positioning.

### Rotate the MU

With the help of the path symbol, you can also rotate the MU. In the example, the MU is to be displayed rotated by 90° around the x-axis. Select the path symbol (click). Select in the context menu of the path symbol: Edit 3D properties. Insert here the rotation (Fig. 11.37).



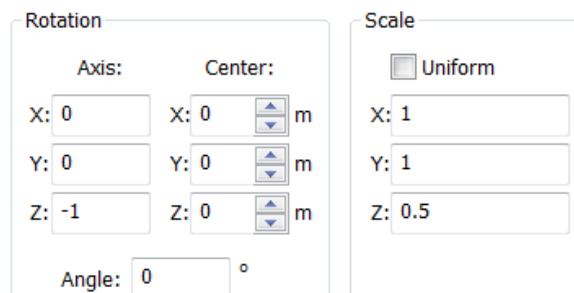
**Fig. 11.37 MU Rotation**

#### 11.2.4 3D State Icons (LEDs)

From Version 11 TR3 onward, Plant Simulation supports LEDs even in the 3D view. The visibility of the LEDs are controlled centrally via Home—Icons. In 3D, the visualization of the states occurs via a series of colored cubes. They are editable as separate objects in 3D (move, rotate, scale). If you, for example, want to change the position of the LED object, then you must do this for all relevant objects.

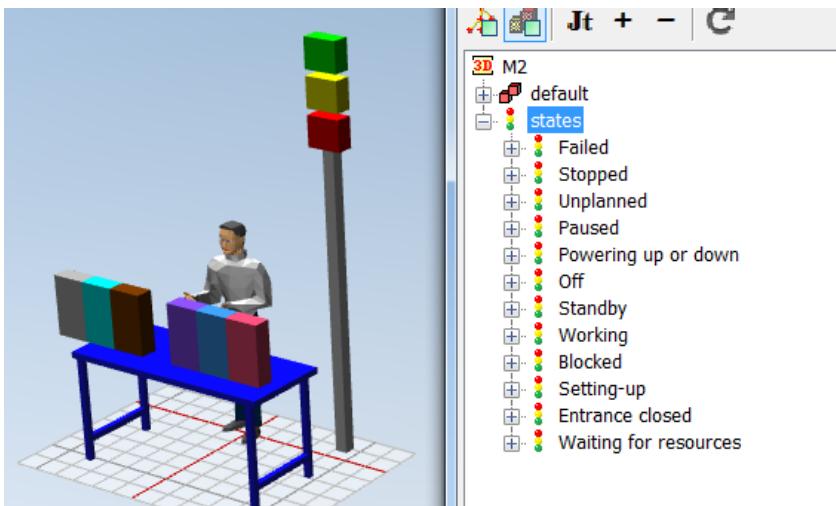
##### Example: Basic 3D (continuation)

The worker should have an Andon-light at his workplace. The Andon-light should show the state of the worker (working, blocked, failed). Open M2 in a new 3D window. First, add a Cuboid (Edit—Cuboid) in the 3D window of M2 (0.1 x 0.1 x 2.5 meters). Right click on the background—Show Graphic Structure. In the graphic structure window, click with the right mouse button on States—Visualize all. Now, all state objects for the object are displayed. Scale the objects for Failed, Blocked and Working (z = - 0.5, see Fig. 11.38).



**Fig. 11.38 Scaling**

Place the state objects like on the top of the "rod" (Fig. 11.39).



**Fig. 11.39** State objects

Note: The icons for "stopped" and "failed" are sharing one position. Therefore, you must first move the icon "stopped" in order to get access to the icon for "failed".

Change the ProcTime of the Drain to two minutes. The state of M2 then changes between "working" and "blocked." For a better visualization, you can move all the state icons to one position.

### 11.2.5 MU Animation

You can animate the MU. The animation starts automatically when the MU enters the block.

#### Example: 3D MU Animation

The MU should rotate during the machining on a vertical machining center. Prepare a frame similar to Fig. 11.40. Use the VerticalDrillingMachine1 from the library Machines3D.



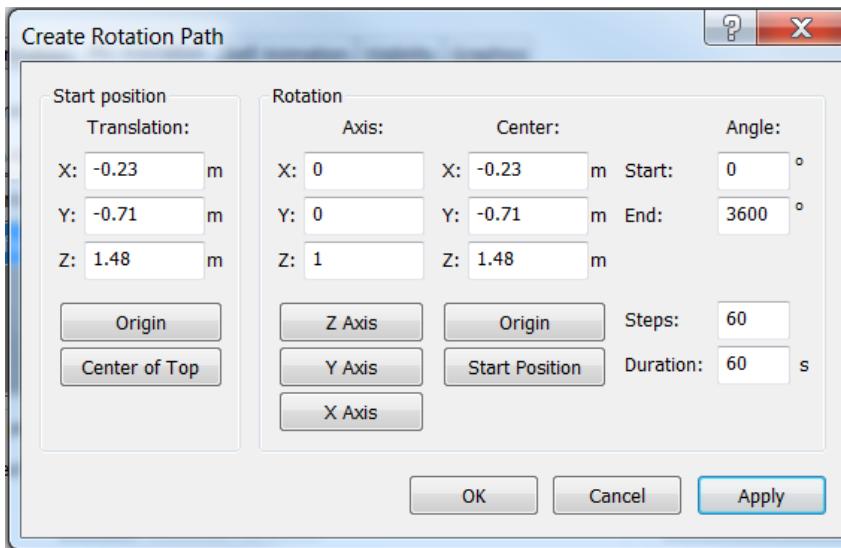
**Fig. 11.40** Example frame

The source generates one MU every three minutes. The processing time of the machine is one minute.

The part should rotate in the processing time 10x ( $3,600^\circ$ ). To do this you only need to create an MU animation. Open the machine in a new 3D window. Right click in the background of the 3D window—Edit 3D properties—Tab MU animation.

Plant Simulation plays the standard animation when the MU enters the block. For most blocks, this is the animation with the name "Default." If you want to change the default behavior, you must replace this animation.

Delete from the tab MU animation the animation path "Default". Then click on Create Rotation Path. Name the new path "Default." Prepare settings as in Fig. 11.41.



**Fig. 11.41** Animation path

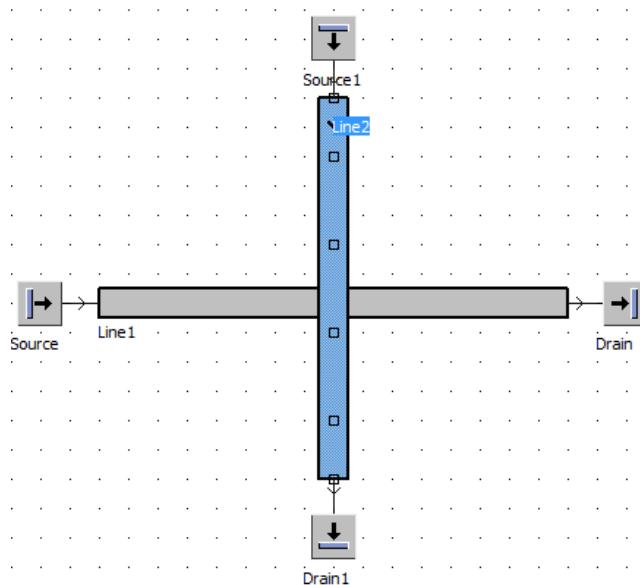
Now, the MU rotates while it is locating on the machine.

### 11.2.6 Length-Oriented Objects in 3D

For length-oriented objects, Plant Simulation generates 3D extrusions. Hence, you can relatively easily change the look of these objects in 3D.

#### Example: Line 3D

Create a frame as in Fig. 11.42. Create also the anchor points as shown in the figure.



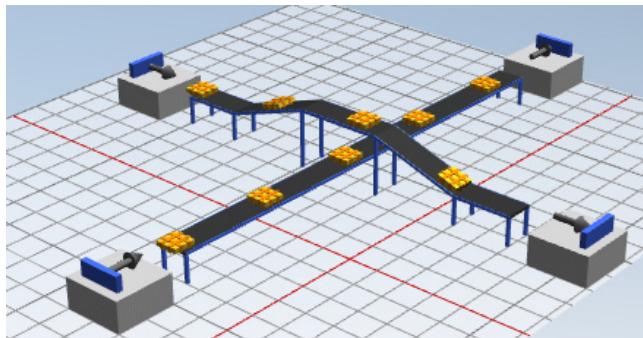
**Fig. 11.42** Example frame

In the dialog of the line, tab Curve—Segment, you can edit the anchor points of the line. In this way, you can also set the z-position. In this example, Line2 is to run at a height of 1.50 meters above Line1. Adjust the Segments table as shown in Fig. 11.43. The coordinates of the points are specified in pixels. Thereby, 40 pixels correspond by default to one meter. 1.50 meters thus corresponds to 60 pixels.

In...	Anchor point X	Anchor point Y	Anchor point Z	Angle	Length	Radius
0	380.000000	100.000000	20.000000	90.000000	40.000000	
1	380.000000	140.000000	20.000000	0.000000	63.245553	
2	380.000000	200.000000	40.000000	0.000000	60.000000	
3	380.000000	260.000000	40.000000	0.000000	63.245553	
4	380.000000	320.000000	20.000000	0.000000	40.000000	
5	380.000000	360.000000	20.000000	0.000000	0.000000	

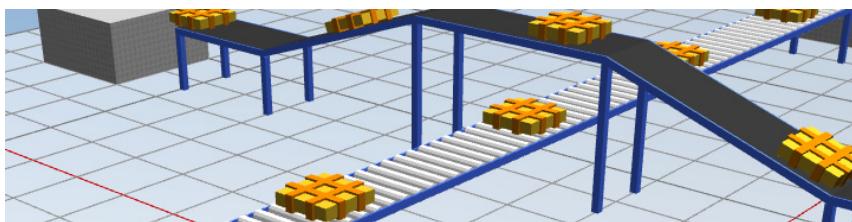
**Fig. 11.43** Segments table

In the 3D view now, Line2 (Fig. 11.44) runs above Line1.



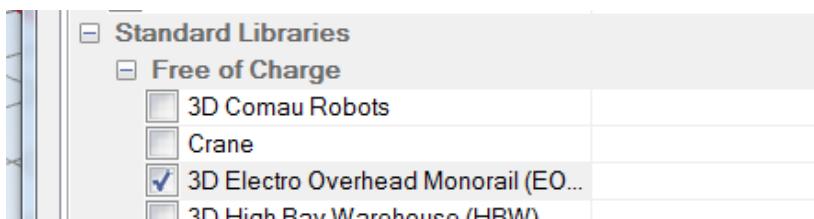
**Fig. 11.44** Example frame

In the 3D properties, you can change the look of the conveyor. In the example, e.g. Line1 should be a roller conveyor. To do this, select from the context menu 3D Properties—Extrusion. Select as type: Roller conveyor with a roller diameter of 0.15 m (Fig. 11.45).



**Fig. 11.45** Roller conveyor

For the modeling of overhead monorail systems in 3D there is a special library in Plant Simulation (EOM, Fig. 11.46).



**Fig. 11.46** Library EOM

### 11.2.7 Textured Plate

You can use a 2D layout to make the 3D model appear more realistic and use the textured plate for positioning the blocks in 3D. Here, the background of the 2D

simulation and the textured plate should have the same scaling. You could do the following. First load the CAD layout into the 2D frame (e.g. drag and drop). Switch to the icon editor for the frame. Select the background image. Select Edit—Export (Fig. 11.47).

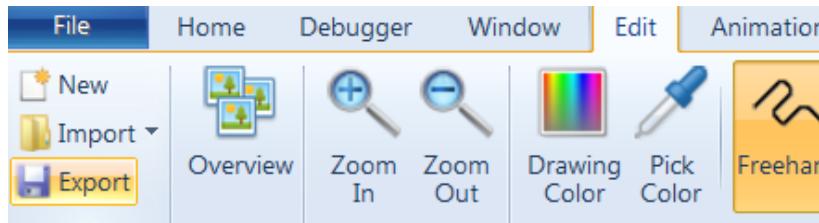


Fig. 11.47 Export background image

Save the image as a PNG file. In 3D, you will find the textured plate in the menu Edit—Facet (Fig. 11.48).

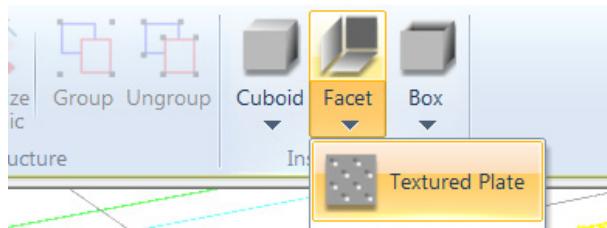


Fig. 11.48 Textured Plate

You must specify the image file and the size of the layout (Fig. 11.49).

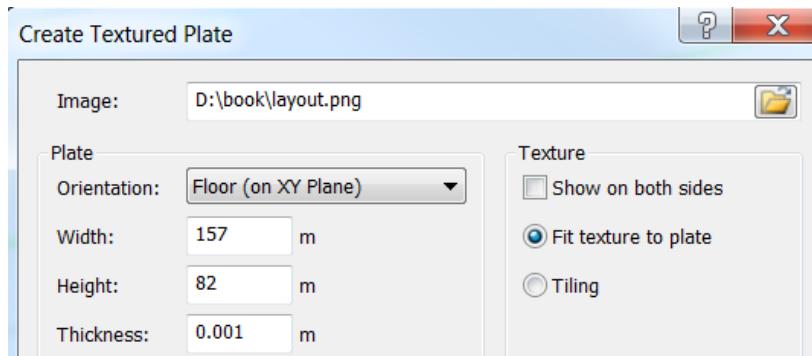


Fig. 11.49 Setup of Textured Plate

You can also add text, fences, walls, etc. The textured plate can be used to display logos on walls.

## 11.3 3D Animation

### 11.3.1 Self-animations, Named Animations

Self-animations manipulate the object itself. For this purpose, prepare an animation and then start it at an appropriate moment. You can move the object on a path or rotate it.

#### Example: 3D self-animation

Create a simple frame (Fig. 11.50).

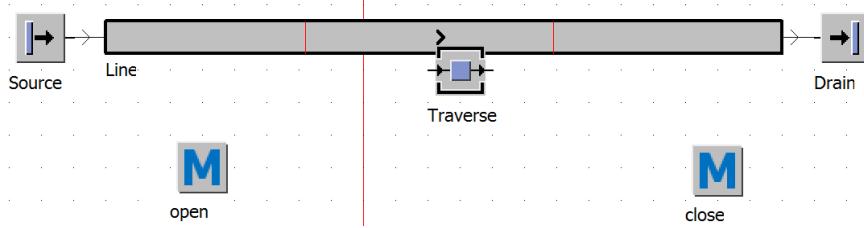


Fig. 11.50 Example frame

Ensure the following settings: The source produces MUs at an interval of one minute; the line has a speed of 0.2 m/sec. Insert two sensors into the line: 6 m (control: open), 14 m (control: close). Activate the 3D view. Open Traverse in a new 3D window. Delete the 3D icon of traverse and insert a Cuboid (3 x 3 x 0.1 meters). Align the Cuboid as in Fig. 11.51.

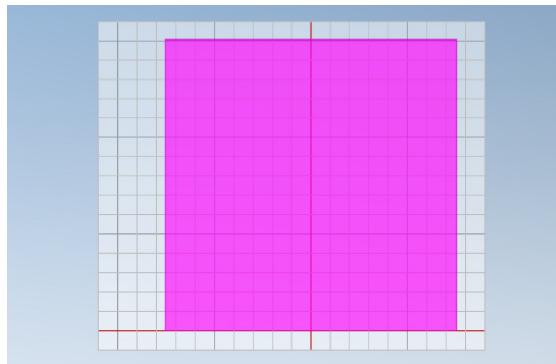


Fig. 11.51 Cuboid

Open the 3D properties of Traverse (right click on the background of the 3D window). Click in Self Animation the button Create Rotation Path. Enter the name "open" and confirm with OK. In the example, the Traverse should pivot 90° around the x-axis upward when passing an MU. Therefore, settings as in Fig. 11.52 are necessary.

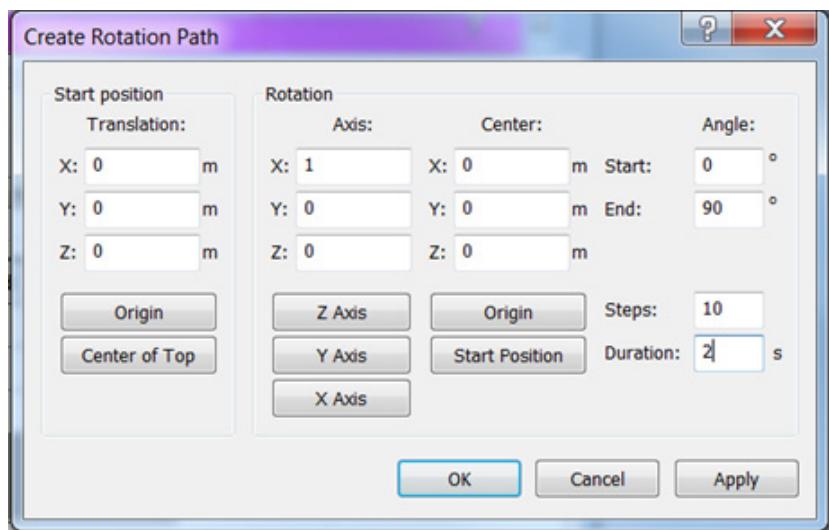


Fig. 11.52 Setting the rotation path

Prepare the second animation analogously to "open," but start with 90° and end with 0°. Name the animation "close." The animation is started with two commands: With `schedule`, the animation is prepared and with `playAnimation`, the animation is played. The open method looks like the following:

```
(SensorID : integer; Front : boolean)
is
do
  traverse._3D.selfAnimations.close.schedule;
  traverse._3D.selfAnimations.playAnimation;
end;
```

Method close:

```
(SensorID : integer; Front : boolean)
is
do
  traverse._3D.selfAnimations.close.schedule;
  traverse._3D.selfAnimations.playAnimation;
end;
```

### 11.3.2 SimTalk 3D Animations, Unnamed Animations

SimTalk provides methods to start animations. To run an animation as in the previous example, the following commands would be necessary:

```
<object>._3D.SelfAnimations.scheduleRotation(
    <startAngle>, <endAngle>, <angleVelocity>);
<object>._3D.SelfAnimations.playAnimation;
```

### Example: 3D Self-animation (continuation)

Change the method open as follows:

```
(SensorID : integer; Front : boolean)
is
do
    traverse._3D.selfAnimations.scheduleRotation(
        0,-90,9);
    traverse._3D.selfAnimations.playAnimation;
end;
```

Method close:

```
(SensorID : integer; Front : boolean)
is
do
    traverse._3D.selfAnimations.scheduleRotation(
        -90,0,9);
    traverse._3D.selfAnimations.playAnimation;
end;
```

### 11.3.3 Camera Animations

You can establish the camera animation along an animation path or attached to an MU. In this way, you can "fly through" the model along the path of the MU. You can switch between the so-called main camera and the animated camera.

#### 11.3.3.1 Attach Camera

##### Example: Camera 3D

Create a simple frame like in Fig. 11.53.

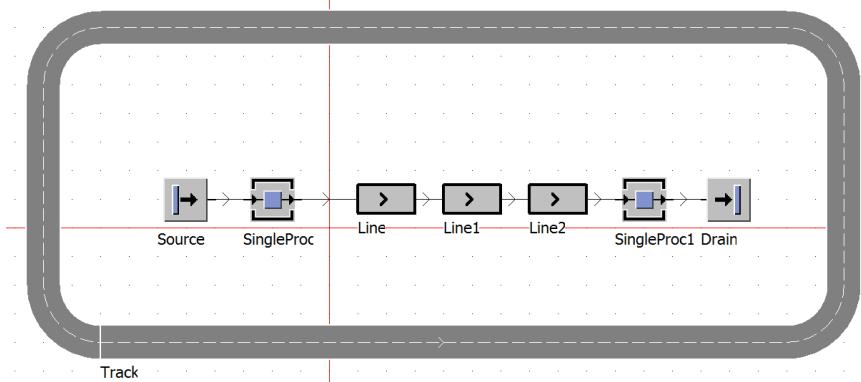


Fig. 11.53 Example frame

Connect the exit of the track with its entrance. One possibility to animate the camera is to "load" the camera onto the MU. Hence, you can travel through the simulation model from the perspective of the moving MU. Especially suitable are transporters that stay for a while in the simulation. Right click on the object and select from the context menu Attach Camera (Fig. 11.54).

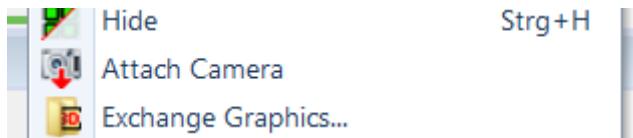


Fig. 11.54 Attach Camera

In the menu View, you find the button Detach to remove the camera from the MU (Fig. 11.55).

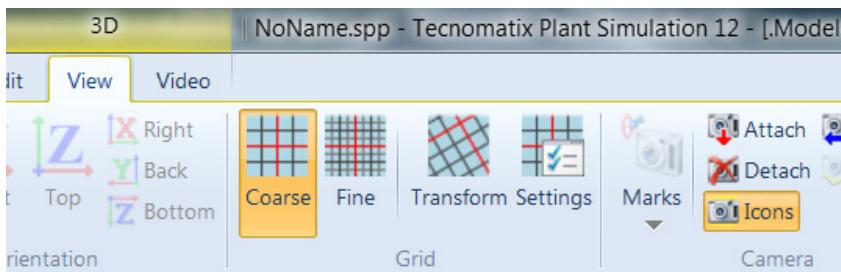


Fig. 11.55 Detach Camera

### 11.3.3.2 Camera Path Animations

You can set up a path and let the camera "fly" on this path. In this way, you can present the model dynamically from different perspectives. For this purpose, you must set up a camera path for the frame. Click with the right mouse button in the background of the frame and open the 3D properties. In the tab Camera Animation, click Add (Fig. 11.56).

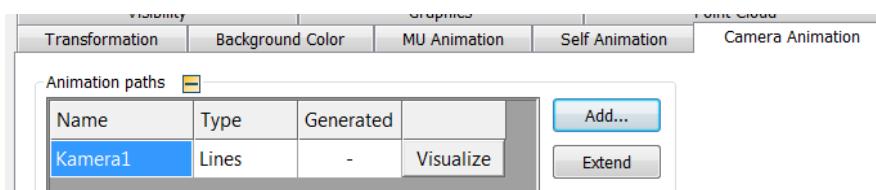
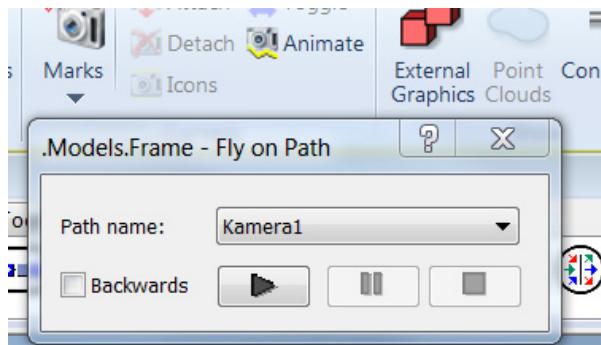


Fig. 11.56 Camera Animation

Enter a name for the animation. Now you need to add the individual camera positions to the animation path. Adjust in your frame the viewpoint (rotate, zoom, etc., and let the 3D properties window open). Then select the animation path and

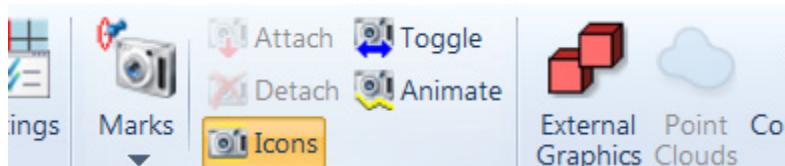
select 'Edit'. It opens a window with the points of the path. Click Add (a new point is created). Highlight the new entry and click Capture Current View. With Edit Values, you can open an editor window for the point on the camera path. There, you can enter a speed or a time for the path segment. Confirm your entry with OK. Repeat this until you have saved all points for the camera path. The camera animation can be started in the menu View—Animate (Fig. 11.57).



**Fig. 11.57** Animate Camera

This opens a dialog box in which you can start and stop the animation. Note: First start the path animation, and then the simulation.

You can launch the attached camera animation and the animation of the main camera in parallel and then switch between the cameras. To do this, click in the menu View on Toggle (Fig. 11.58).



**Fig. 11.58** Toggle between Cameras

#### 11.3.4 Manipulation of 3D Objects

With the help of SimTalk, you can manipulate 3D objects during the runtime of the experiments. You can, for example, change the assigned material or the visibility of objects.

##### Example: 3D Worker Utilization

In the 3D representation of the simulation, the utilization of workers should be dynamically represented by different colors of the shirts of the workers. Create a frame as in Fig. 11.59.

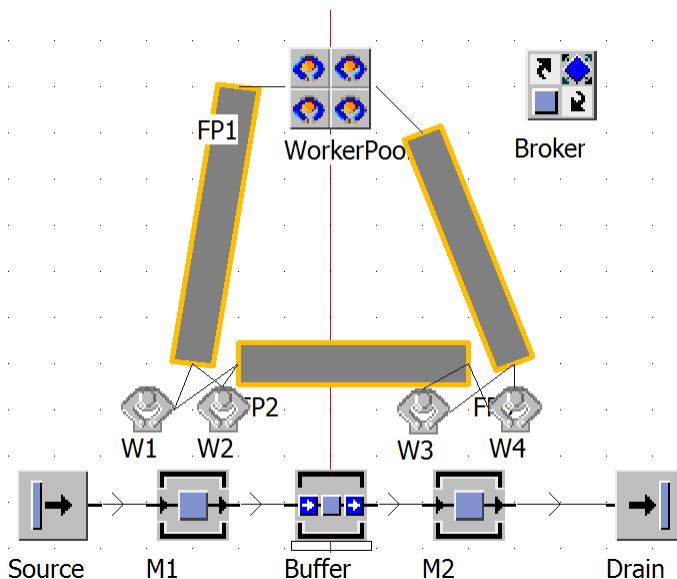


Fig. 11.59 Example frame

Ensure the following settings: The source generates one part every 12 minutes. M1 has a processing time of 10 minutes, 75 per cent availability and 10 minutes MTTR. Buffer has a capacity of 100 parts. M2 has a processing time of five minutes, an availability of 65 per cent and 10 minutes MTTR. Activate in M1 and M2 the Importer and the Failure Importer. Set in the Failure Importer as service "repair." The WorkerPool generates three workers, of which one worker has the additional service "repair."

The addressing of 3D graphics is done using an array in which you must specify the location of the graphic. You find the position of the graphic in the graphic structure of the object. Open the worker from the class library in a 3D window. Right click on the background of the 3D window and select Show Graphic Structure (Fig. 11.60).

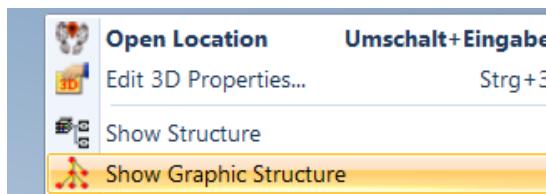
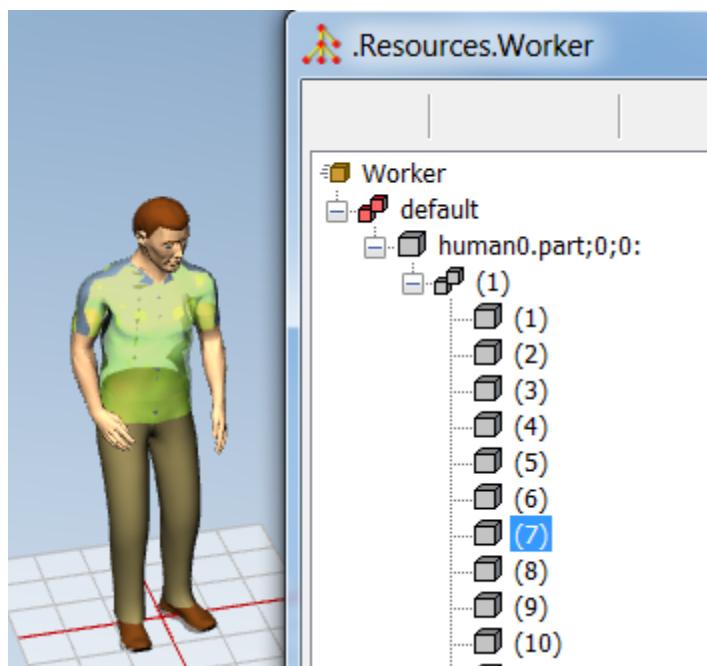


Fig. 11.60 Show Graphic Structure

You can click on the individual elements in the graphic structure. The corresponding graphic is highlighted in the 3D object in green. The shirt of the worker is in the symbol "default" in the first folder, first subfolder, position 7 (1,1,7), as in Fig. 11.61.



**Fig. 11.61** Graphic structure

The color of the graphic can be set with the command: `<path>._3D.setGraphicMaterial`. In order to trigger the color change, you can use an observer for the attribute `AvailableForMediation` (version 12). The attribute changes when the worker becomes available and if the worker was mediated. Insert into the worker in the class library an appropriate observer. Create the method with F4 as an internal method in the worker. The method could look like this:

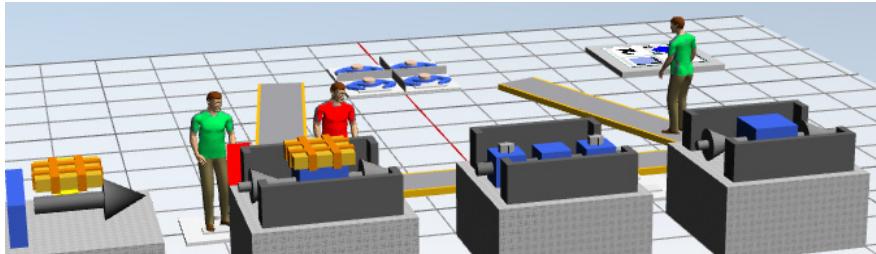
```
(attribute: string; oldValue: any)
is
  color:integer;
do
  - The method is also called if worker is deleted
  if to_str(self.~) /= ".<DELETED>" then
    if (self.~.statServicesRepairingPortion+
        self.~.statServicesWorkingPortion) > 0.9 then
      color:=makeRGBValue(255, 0, 0);
    elseif (self.~.statServicesRepairingPortion+
            self.~.statServicesWorkingPortion) > 0.7 and
            (self.~.statServicesRepairingPortion+
             self.~.statServicesWorkingPortion) <= 0.9 then
      color:=makeRGBValue(255, 255, 87);
    else
```

```

        color:=makeRGBValue(0, 192, 75);
    end;
    self.~._3D.setGraphicMaterial(makeArray(1,1,7),
        color, color,0,0,0, 0.1);
    end;
end;

```

If the worker has capacity utilization of more than 90 per cent, his shirt is shown in red; green is for below 70 per cent, and yellow for between the two ranges (Fig. 11.62).

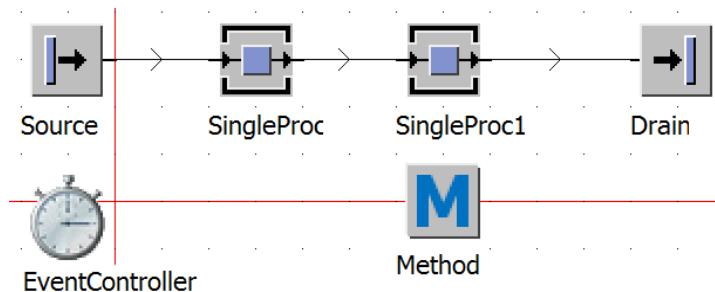


**Fig. 11.62** Visualization of utilization

Changing the visibility of graphics can help, to visualize different processing states of MUs. To do this, you could insert different graphic groups into the 3D object and change the visibility dynamically.

#### Example: 3D Processing States

Create a frame according to Fig. 11.63.



**Fig. 11.63** Example Frame

Method is the entrance control of SingleProc1. Open the Entity in 3D. Right click in the background and show the graphic structure. Right click on Entity and select from the context menu: New Graphic Group. Name the new graphic group as “proc1”. You can show and hide graphic groups using the context menu. Hide the graphic group default and set the graphic group proc1 to visible. Insert a cylinder into the graphic group proc1 (Edit-Cuboid-Cylinder, Fig. 11.64).

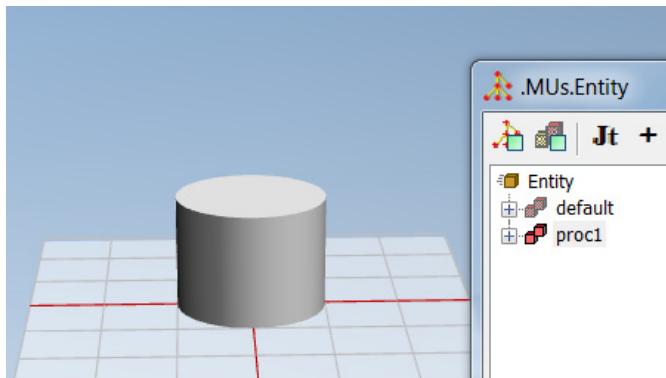


Fig. 11.64 Graphic group proc1

Hide proc1 and set default to visible. You can use the command `hideGraphicGroup` to hide and `showGraphicGroup` to show graphic groups. For changing the 3D icon you need to hide the “old” icon and show the new icon. Method:

```
is
do
  @_3D.hideGraphicGroup("default");
  @_3D.showGraphicGroup("proc1");
end;
```

# Chapter 12

## Integrate Energy Consumption and Costs

Since version 11, Plant Simulation objects have included attributes for mapping energy consumption. In addition, it is possible to expand the building blocks by one's own attributes. Thereby, one can include aspects into the simulation that are not set in Plant Simulation by default.

### 12.1 Simulation of Energy Consumption

In the case of material flow elements, you can define energy consumption values for different operating states. This can be used to run experiments in which specific energy states are changed. You can also access the energy attributes using SimTalk in order to represent more complex behavior.

#### 12.1.1 Energy Consumption—Basic Behavior

For active material flow elements, a number of energy states are defined: working, setting up, operational, failed, standby and off. You can assign to each of these states a power consumption value in the dialog boxes of the objects. If the status of the block changes, the current energy consumption value is adjusted by Plant Simulation. Plant Simulation records the sum of the consumed energy for each block with an active energy function. Plant Simulation provides the EnergyAnalyzer as a statistical tool for evaluating energy consumption.

#### Example: Basics of energy consumption

Create a simple frame as in Fig. 12.1.

Prepare the following settings: M1 and M3 each have a processing time of one minute. M2 has a processing time of 90 seconds. The source generates MUs at intervals of two minutes (non-blocking). All SingleProcs have an availability of 90 per cent and an MTTR of 10 minutes. The ShiftCalendar retains its default settings. Assign the ShiftCalendar to all SingleProcs and the source. Set the end of the EventController to 100 days. Activate in all SingleProcs the energy functions and set the values as in Fig. 12.2.

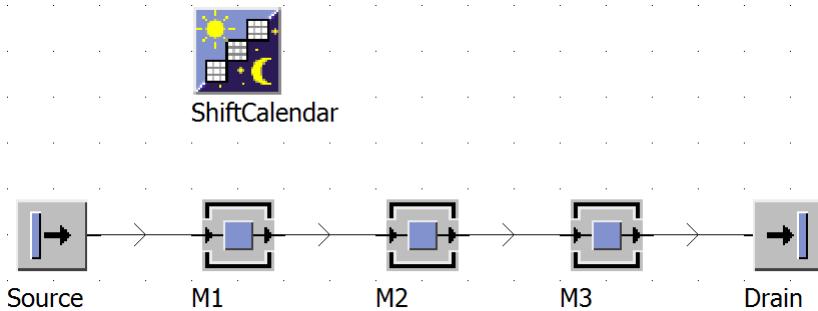


Fig. 12.1 Example frame

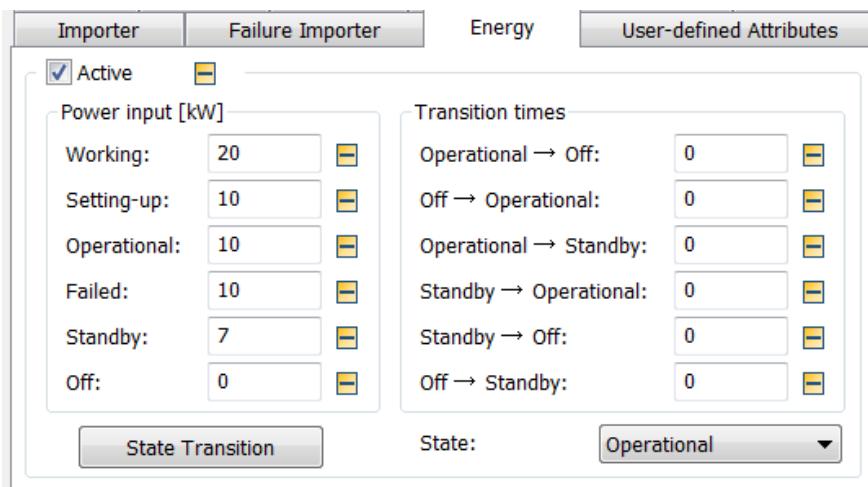


Fig. 12.2 Energy function

### Transition times

The transition times determines the temporal distance between energy states. This time, the block remains in the old energy state. Subsequently, the new energy state is established. Click on State Transition. Here, you determine which energy states are associated with the paused and unplanned states. This means that during breaks and unplanned time, the machines remain in the standby mode (Fig. 12.3).

"Power up early" means that, at the end of a pause, the change in the energy state starts early, taking into account the transition times when the target energy state is reached by the end of the pause (i.e. no time is lost due to "power up"). Let the simulation run for 100 days. Then, open a SingleProc to the tab Statistics. Click on Energy Statistics. The block has its own energy statistics (Fig. 12.4).

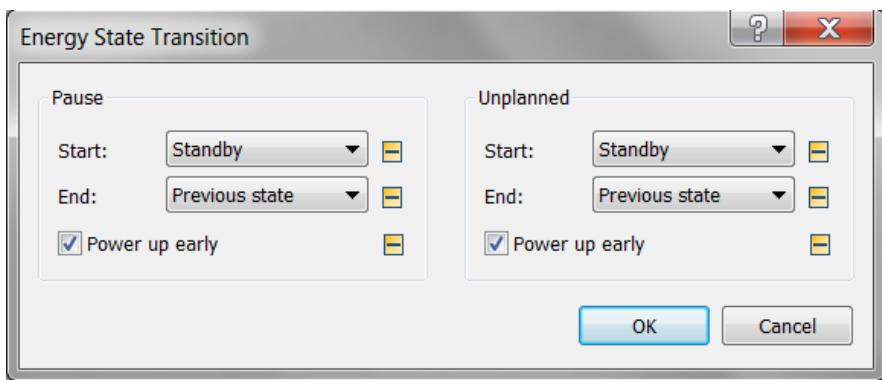


Fig. 12.3 Energy State Transition

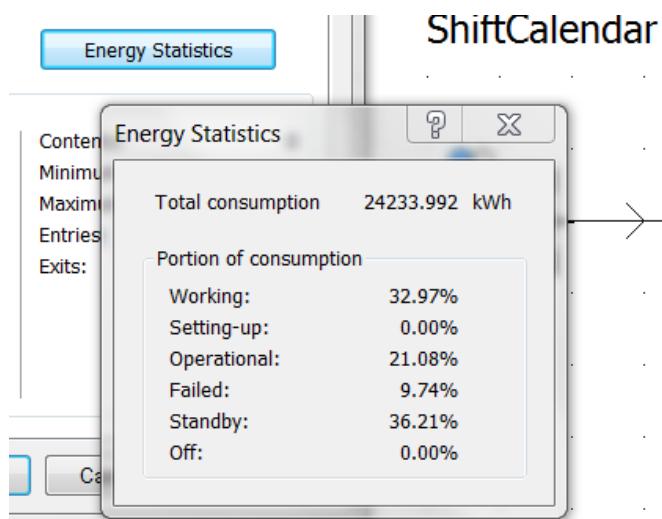


Fig. 12.4 Energy Statistics

Should you decide to switch off the stations during the unplanned time, change the State Transition of M1 as in Fig. 12.5.

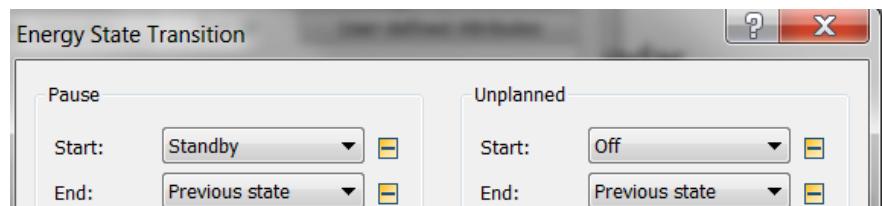
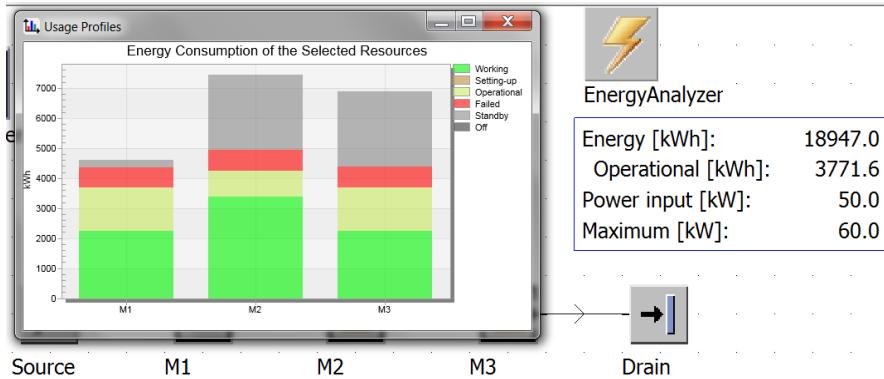


Fig. 12.5 State Transition

Let the simulation once again run for 100 days and compare the results. Drag an EnergyAnalyzer into your frame (class library—folder Tools). Open the EnergyAnalyzer to the tab Objects. Click on "Add all." The EnergyAnalyzer performs some evaluations—e.g. a prepared display panel (Settings—Display panel, Fig. 12.6).



**Fig. 12.6** EnergyAnalyzer

### 12.1.2 Energy Profiles

A number of equipment has no constant energy consumption, for example, during the processing time. In such cases, you have to change the power consumption during the processing time according to a recipe or a work plan. Analogously, you can also model inrush peaks. The attributes for the power input starts with PowerInput. The attribute to control the power input during operation of the block is PowerInputWorking.

#### Example: Energy simulation chamber furnace

In a production, a chamber furnace for the thermal treatment of parts is used. The treatment lasts three hours. In the first hour, the oven has a power input of 30kW, in the second hour 10kW, and in the third hour one kW. The charge (refill) of the furnace takes 30 minutes. During this time, no energy is consumed. Create a simple frame as shown in Fig. 12.7.

Ensure the following settings: The source produces parts at intervals of 3:30:00. The Oven has a processing time of three hours and a set-up time of 30 minutes. Activate the Energy function. Set all power input values to zero. Enter the SetPower method as an input control (before actions) in the Oven. Activate the plotter of the EnergyAnalyzer. Create the table PowerProfile as in Fig. 12.8.

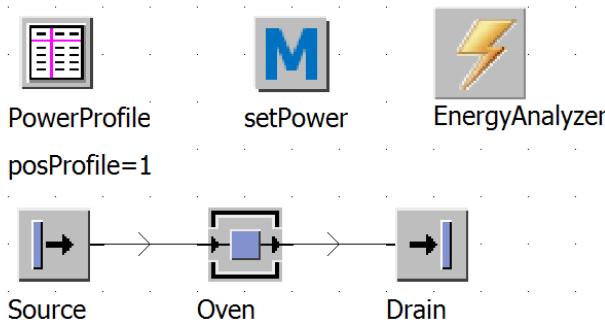


Fig. 12.7 Example frame

	time 1	real 2
string	time	power
1	0.0000	30.00
2	1:30:00.0000	10.00
3	2:30:00.0000	1.00

Fig. 12.8 Table PowerProfile

The first column contains relative times (starting from the entrance of the MU into the block). The second column contains the current that needs to be set. The method SetPower sets the value of powerInputWorking and calls itself at the next time in the table. This might look like this:

```

is
do
  --set power for state working
  Oven.powerInputWorking:=PowerProfile[2,posProfile];
  -- call the method again, after a distance
  if posProfile<powerProfile.yDim then
    self.methcall(PowerProfile[1,posProfile+1]-
      PowerProfile[1,posProfile]);
    posProfile:=posProfile+1;
  else
    posProfile:=1;
  end;
end;

```

The energy consumption now follows the given profile (Fig. 12.9).

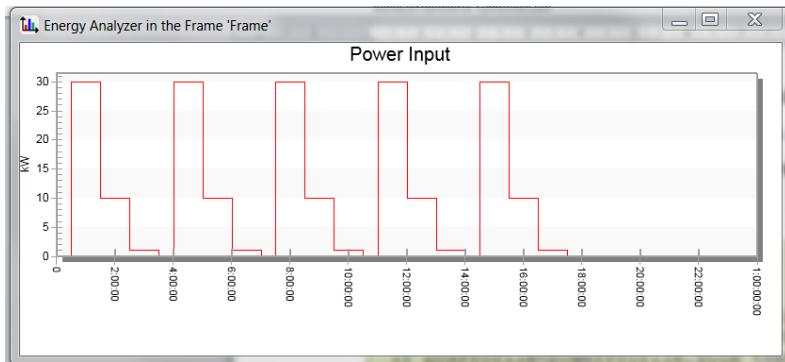


Fig. 12.9 Energy profile

### 12.1.3 Energy Consumption in the Periphery of Machines

A significant portion of energy consumption occurs in the periphery of machinery and equipment. This might include, for example, lighting, ventilation, cooling, and the like. Only some of these consumers have a direct connection to the production process and, usually, have their own trigger for changing states. Nevertheless, you can use the energy functions of Plant Simulation for modeling these consumers. This works quite well with dummy objects in which you manipulate the energy consumption in the state "operational." Thereby, you can create a network of energy consumers.

#### Example: Energy simulation periphery

You are intended to simulate the energy consumption of the factory lighting. Create a simple network as in Fig. 12.10.

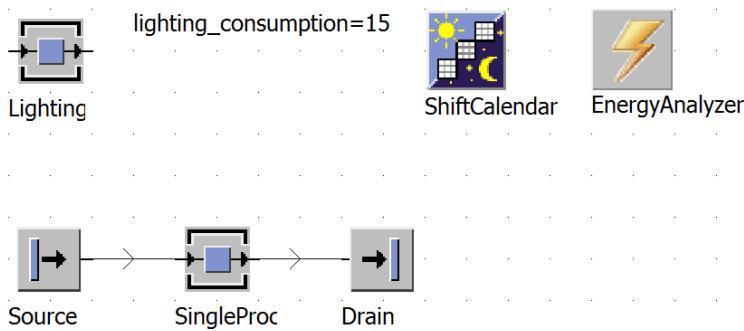


Fig. 12.10 Example frame

Let all blocks retain their default settings. Enable the energy functions of the block Lighting. Set all power input values to zero. In the non-working

time (unplanned), the lighting is switched off. While working, the power consumption of lighting is 15 kW. Insert an observer for the attribute unplanned in the ShiftCalendar object. The observer method sets the powerInputOperational of lighting and should look like this:

```
(attribute: string; oldValue: any)
is
do
  if oldValue=false then
    --start of unplanned light off
    lighting.powerInputOperational:=0;
  else
    --end of unplanned, light on
    lighting.powerInputOperational:=
      lighting_consumption;
  end;
end;
```

In the same way, you could decrease the lighting in the pauses by 50 per cent (observer in the ShiftCalendar for the attribute pause):

```
(attribute: string; oldValue: any)
is
do
  if oldValue=false then
    --reduce lighting by 50% in pauses
    lighting.powerInputOperational:=
      lighting_consumption/2;
  else
    lighting.powerInputOperational:=
      lighting_consumption;
  end;
end;
```

This results in energy consumption for lighting as in Fig. 12.11.

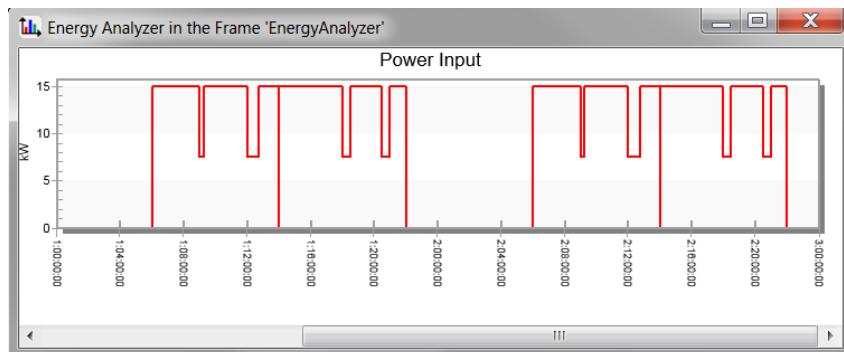


Fig. 12.11 Lighting

## 12.2 Integrate Costs into the Simulation

Using custom attributes, costs can be easily taken into account in the simulation.

### 12.2.1 Production Concurrent Costing

For many aspects, it is interesting how much production cost a product has already incurred up to a certain production level. Production costs are the basis for calculating fixed capital within a production process (e.g. current assets, inventory). The target of many improvement measures is to reduce fixed capital. For determining manufacturing costs, different methods of calculation exist. A possible calculation is:

- Manufacturing materials
- + Manufacturing wages
- + Special direct costs of production
- = Minimum production costs
- + General expense for materials
- + General expense for production
- + General expense for administration
- + Interest on debt capital
- = Highest production costs

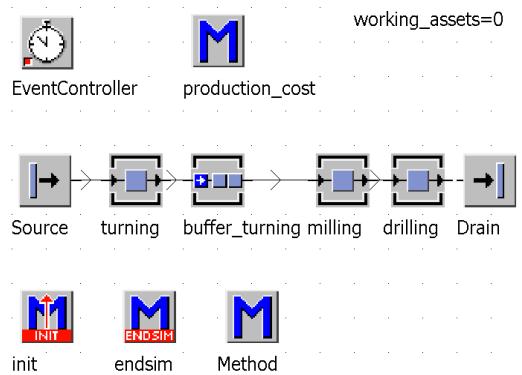
The general expense is calculated in this manner:

direct expenses \* cost rate. The following simple approach should be chosen for the simulation:

The basis for determining the production costs are the costs of materials and direct manufacturing costs. Direct production costs are calculated using the time multiplied by the hourly wage or the hourly machine rate. The individual parts must now collect this information during processing (simulation). One way to realize this is by employing user-defined attributes.

#### Example: Production costs and working assets

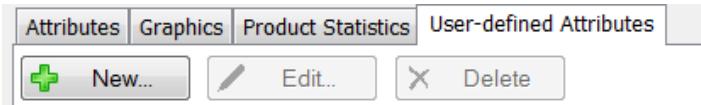
Create a frame like in Fig. 12.12.



**Fig. 12.12** Example frame

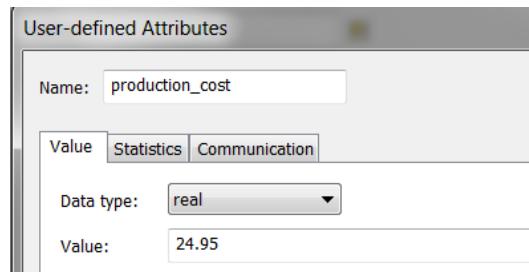
Material costs of the unfinished part: €24.95; average manufacturing wage: €/h36.

The part is first turned, then milled, and then drilled. The processing time for turning is one minute; that for milling and drilling is one minute as well. The relevant entity (Part) is to have the property "production\_cost" (data type real). Procedure: Duplicate an entity. Rename it as "part." Open the entity by double-clicking it in the class library. Select User-defined attributes—New (Fig. 12.13).



**Fig. 12.13** User-defined Attributes

Enter the following values from Fig. 12.14 in the dialog.



**Fig. 12.14** User-defined Attribute

The value of the attribute is initialized with the value of the raw material (at the beginning of processing). The machines need to have an attribute "wage" (data type real), into which the hourly wage costs are entered. The calculation of the production costs is relatively simple (e.g. exit control rear of the machines, method production\_cost):

```
is
do
  @.production_cost:=
  @.production_cost+((?.procTime/3600)*(?.wage));
  -- procTime in seconds!
end;
```

**Note:** The anonymous identifier ? returns a reference to the object whose control was triggered.

Even in assembly operations, the manufacturing costs of the parts can be combined and transferred to the new part (if you destroy the individual parts).

## 12.2.2 Working Assets

To determine the working assets, you have to identify the existing entities and their cost (ideally within an EndSim method). There are a number of approaches for this.

### Example: Production costs and working assets—continuation

The value (costs) of the parts in the frame must be determined after the end of the simulation. For this purpose, you query the individual objects regardless of whether they are occupied or not. If they are occupied, then the costs of the parts are added to a global variable (working\_assets). Method endSim:

```

is
  i:integer;
do
  if turning.occupied then
    working_assets:=
    working_assets+turning.cont.production_cost;
  end;
  if milling.occupied then
    working_assets:=
    working_assets+milling.cont.production_cost;
  end;
  if drilling.occupied then
    working_assets:=
    working_assets+drilling.cont.production_cost;
  end;
  --query each place individually
  if Buffer_turning.occupied then
    from i:=1; until i=Buffer_turning.capacity loop
      if Buffer_turning.pe(i).cont /= void then
        working_assets:=working_assets+
        Buffer_turning.pe(i).cont.production_cost;
      end;
      i:=i+1;
    end;
  end;
end;

```

For a large number of objects, this approach is very involved. With the frame object, you can access all objects in the frame. With `Frame.numNodes`, you can determine the number of all objects in the frame. The individual objects can be accessed with `Frame.node(index)`. With the method `class`, you can check the type of an object. A universal method for calculating the working assets could look like this:

```

is
  i:integer;
  k:integer;
do
  working_assets:=0;
  for i:=1 to current.numNodes loop
    if current.node(i).class.name="SingleProc" or
      current.node(i).class.name="PlaceBuffer" or
      current.node(i).class.name="Source"
      -- and so on
    then
      for k:=1 to current.node(i).numMu loop
        working_assets:=working_assets+
          current.node(i).mu(k).production_cost;
      next;
    end;
  next;
end;

```

Similarly, you could also calculate weights (data type weight), among other things.

Another possibility is the use of inheritance for determining the MUs (e.g. entities). The class knows, at any given time, how many derived objects exist in the simulation (child objects). In this way, you can collect all the necessary information about the child objects with the help of the class (parent). You need the following SimTalk methods (Table 12.1):

**Table 121** SimTalk Methods

Method	Description
<path>.numChildren	Returns the number of instances of the class in the simulation (in all frames)
<path>.childNo(<integer>)	Returns a reference to the instance of the class with the index <integer>

The calculation of the working asset is easiest if you have only one frame in the file or all frames are part of one simulation model (no options and alternatives in the same file). You must address this starting at the class in the class library. The method of determining the current assets would then look like this:

```

is
  i:integer;
do
  working_assets:=0;
  -- for all objects of the class part
  for i:=1 to .MUs.Part.numChildren loop
    -- add costs of the parts

```

```

working_assets:=working_assets +
    .MUS.Part.childNo(i).production_cost;
next;
end;

```

### 12.2.3 Machine-Hour Rates

Calculating the machine-hour rate allows a more accurate allocation of common costs and, thus, a more accurate calculation. The goal is the apportionment of the machine-related indirect production costs to one hour of machine running time. Calculating the machine-hour rate consists of the following components (sample):

Machine-dependent indirect production costs	Fixed amount per month	Variable costs per hour
1. Imputed depreciation <sup>3</sup>	4,500	
2. Imputed interest <sup>4</sup>	900	
3. Imputed rent <sup>5</sup>	500	
4. Energy costs per hour		0.25
5. Tooling costs		10.00
6. Repair/Maintenance		10.00
7. Fuel costs		2.50
Indirect production cost per hour	5,900/number of hours of machine running time	22.75
Machine-hour rate	5,900/number of hours per month + per month + 22.75	

You can calculate the machine-hour rate in a TableFile. First, create a sample of the machine-hour rate calculation in the class library (the calculation scheme is the same for all machines, only the values will change). Create the table according to Fig. 12.15.

	string 0	real 1
string		Values
1	Replacement value	2000000.00
2	asset depreciation range in months	120.00

Fig. 12.15 Table Machine\_Hour\_Rate

<sup>3</sup> Replacement value/asset depreciation range in months.

<sup>4</sup> Cost value/2 \* imputed rate/100.

<sup>5</sup> Footprint of the machine in m<sup>2</sup> \* imputed rent per month.

### Calculations in tables

You can enter values directly into the table cells. Alternatively, you can specify formulas that calculate the values in the table cells. Therefore, tables and lists have two modes: In the formula mode, you can enter formulas, while in the input mode you enter values and the values of the formulas are displayed. You can switch to the formula mode using the formula button (to the left of Open, Fig. 12.16).

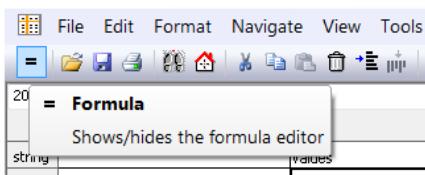


Fig. 12.16 Formula mode

The calculated fields are shown with a light blue background. A formula has the following basic structure:

PathTableFile[c,r] operator PathTableFile[c,r]

This is somewhat cumbersome in relation to the same table; hence, in calculations within a table, you can use the anonymous identifier "?" as a substitute for the path.

E.g. the imputed depreciation cost is calculated in the following way:

Replacement value/asset depreciation range in months

As a formula in the table in the example above, you would enter:

?[1,1]/?[1,2]

If your formula is wrong, Plant Simulation shows an error message (Fig. 12.17).

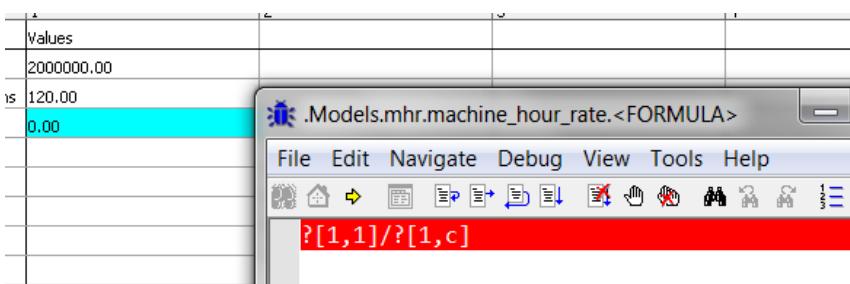


Fig. 12.17 Error message

It is important to consider the data types in calculations. The result cell has a certain data type (determined by the data type of the column in the table). The result must also have this data type; otherwise, Plant Simulation will show an error message. A reasonable simplification is the calculation in a single data type (e.g. real) and formatting of the output (e.g. money). The calculation of the machine-hour rate results in the following table (and the associated formulas):

	Column 0	Column 1
1	Replacement value	2000000.00
2	Asset depreciation range in months	120.00
3	Imputed depreciation	?[1,1]/?[1/2]
4	Cost value	1500000.00
5	Imputed rate (%)	7.00
6	Imputed rent	?[1,4]*?[1,5]/200
7	Footprint of the machine in m <sup>2</sup>	20.00
8	Imputed rent per month and m <sup>2</sup>	13.00
9	Imputed rent per month	?[1,7]*?[1,8]
10	Total fixed costs per month	?[1,3]+?[1,6]+?[1,9]
11		
12	Energy costs per hour	0.25
13	Tooling costs per hour	10.00
14	Repair/Maintenance per hour	10.00
15	Fuel costs per hour	2.50
16	Total indirect production costs per hour	?[1,12]+?[1,13]+?[1,14]+?[1,15]
17		
18	Monthly machine running time in hours	1.00
19		
20	Machine-hour rate	?[1,10]/?[1,18]+?[1,16]

The cell [1.18] must be calculated in the simulation. The result of the simulation is the actual occupancy of the machine and, following the calculation in the table, the machine-hour rate (taking into account breaks, occupancy, maintenance, and whatever else is required in the simulation). Create a frame like in Fig. 12.18.

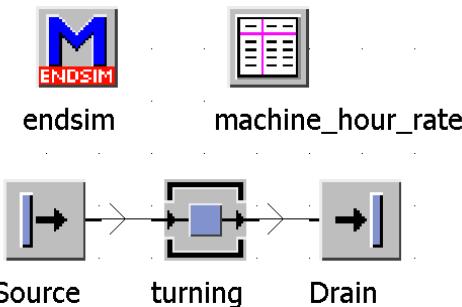


Fig. 12.18 Example frame

At the end of the simulation, the method EndSim reads (set a month as the end of the simulation in the event controller) the working time of the object turning from the statistics data and writes it to the table machine-hour rate (cell [1,18]). Then, the table calculates the machine-hour rate.

Program the method endSim:

```
is
do
  -- set the machine working time in the table
  machine_hour_rate[1,18] :=
    turning.statWorkingTime/3600;
end;
```

**Note:** The values in the table retain their values between the simulation runs. Hence, at the beginning of the simulation, you need to initialize all required values (in the example above, the cell [1,8]).

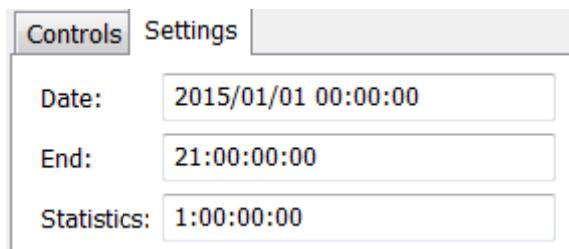
# Chapter 13

## Statistics

For all simulations you have to collect, visualize and evaluate data. Therefore, Plant Simulation provides a wide range of instruments.

### 13.1 Statistics Collection Period

The statistics collection period is the interval between activating the collection of statistical data and the query of statistics. Statistical data are recorded only if the collection of statistics in the objects is active. If statistics is disabled, all statistical data of an object will be deleted. You can reset statistics collection in the Event-Controller at any time. In this way, you can hide the ramp-up behavior of your model, and statistics collection can start when the system reaches full output. You can enter this setting in the EventController on the tab Settings. Enter the time at which the EventController will reset statistics into the field Statistics (Fig. 13.1).



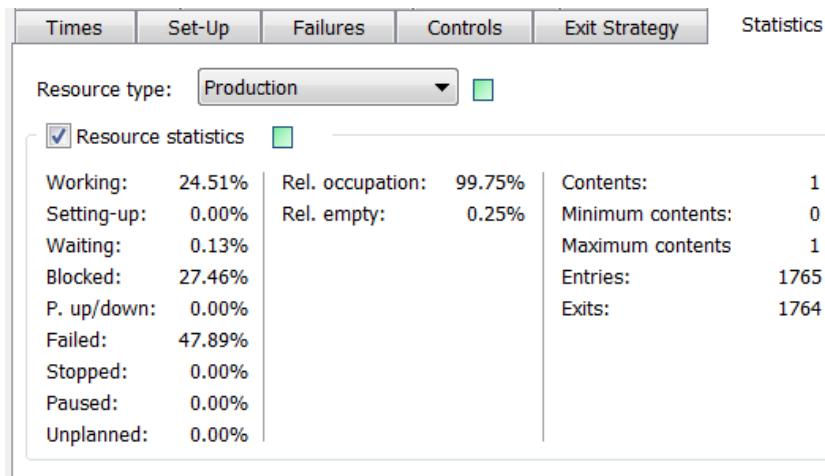
**Fig. 13.1** EventController settings

With the setting above, the EventController will reset statistics of the objects after one day. When the simulation is finished after 11 days, the objects record statistical data for 10 days. The following description illustrates the composition of the statistics collection period.

Statistics collection period				
Planned				Not planned
Not paused				Paused
operational				Not operational
Waiting	Setup	Working	Blocked	Failed/stopped

A resource is unplanned, when you use a shift-calendar for this resource and this time was not planned in the calendar (e.g. weekend or non-working night). The effect of the unplanned state is the same as of the state paused. A resource pauses when its paused attribute has the value true. Work in progress is interrupted by the paused state. You cannot move MUs on a paused object. Within the non-paused time, the resource can be either operational or not.

An object is not operational if it is failed, was stopped by a LockoutZone or is blocked by a "material flow jam." A resource is working if at least one MU is processed on the block (setup times and recovery times are not part of the working time). You find a summary of the most important statistical data for the material flow blocks in its dialogs in the tab Statistics (Fig. 13.2).



**Fig. 13.2** Statistics

P. up/down belongs to the energy functions and reflects the times for power up and down. The Rel. empty value is the time in which no MUs are located on the block while the block was operational.

#### Activation of statistics collection

You can enable or disable the resources statistics of a block using SimTalk:  
`<path>.ResStatOn:=true; - or (false);`

Note: By default, statistics collection is turned on for all material objects. To increase the performance of the simulation, it can help to deactivate statistics collection for all objects for which you do not need statistical analysis.

## 13.2 Statistics—Methods and Attributes

You can read all the statistics data with SimTalk. Most of them begin with "stat" (Table 13.1).

**Table 13.1** SimTalk Statistics

Method	Description
<code>&lt;path&gt;.statistics</code>	Shows statistics of the object on screen
<code>&lt;path&gt;.statistics(&lt;table&gt;)</code>	Statistics will be written in the specified Plant Simulation table
<code>&lt;path&gt;.statistics(&lt;string&gt;)</code>	Statistics will be written in the specified file
<code>&lt;path&gt;.statWaitingPortion</code>	Returns the percentage of the waiting time relative to the total time (data type real)
<code>&lt;path&gt;.statWorkingPortion</code>	See above
<code>&lt;path&gt;.statBlockingPortion</code>	
<code>&lt;path&gt;.statFailPortion</code>	
<code>&lt;path&gt;.statSetupPortion</code>	
<code>&lt;path&gt;.statPausingPortion</code>	
<code>&lt;path&gt;.statUnplannedPortion</code>	
<code>&lt;path&gt;.statEmptyPortion</code>	
<code>&lt;path&gt;.statNumIn</code>	Returns the number of MUs that entered (left) the object. The returned data type is integer.
<code>&lt;path&gt;.statNumOut</code>	
<code>&lt;path&gt;.statMaxNumMU</code>	Returns the maximum number of places occupied during the simulation.
<code>&lt;path&gt;.initStat</code>	The method initStat resets the statistics of an object. This can be useful if the statistics recording should not start at the beginning of the simulation; the statistics collection will contain only values from the call of initStat.

### Example: Statistics

You are to simulate a manufacturing cell. Four machines feed a chemical treatment unit. The chemical treatment unit has a constant feed rate of 0.00333 m/s. The part has a length of 0.4 meters. The machines have a processing time of four minutes, an availability of 50 per cent and an MTTR of three hours. A buffer with a capacity of 100 parts is located in front of the chemical treatment unit, which is 30 meters long. The source produces parts at an interval of two minutes. Create a frame like in Fig. 13.3.

This example will show some typical statistical analysis.

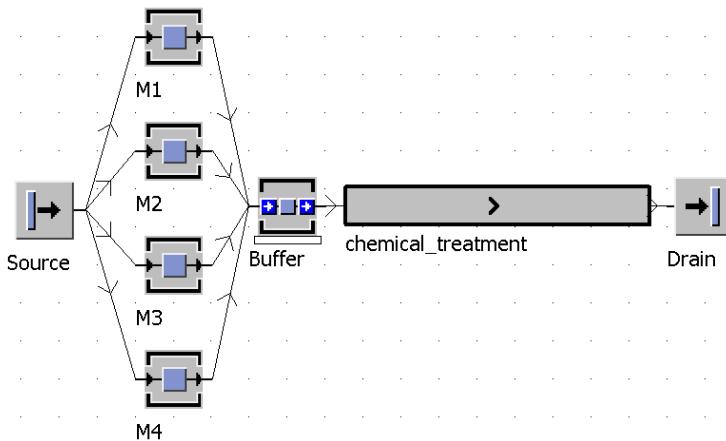


Fig. 13.3 Example frame

### 13.2.1 Write the Statistical Data into a File

The statistical data is to be written into a file at the end of the simulation. This can be easily accomplished with a table. During or at the end of a simulation, you write the statistics data into a table and then save the table as a file. For this, the TableFile provides several methods (Table 13.2).

Table 13.2 Methods for data export

Method	Description
<code>&lt;path&gt;.writeFile(&lt;string&gt;)</code>	This method writes the contents of the table into a text file. Pass the path as argument. Existing files with the same name will be overwritten.
<code>&lt;path&gt;.writeExcelFile(&lt;string&gt;,[&lt;string&gt;])</code>	This method writes the contents of the table into an Excel file. Pass the filename (path). As a second argument, you can pass an Excel table name. To be able to use this method, Microsoft Excel has to be installed on your computer.

To simplify statistical analysis, you should write a method that writes the statistical data of all material flow objects to a TableFile. Add a TableFile “analysis” to the frame. Format the table according to Fig. 13.4.

	string 0	real 1	real 2	real 3	real 4	real 5
string		working	waiting	blocked	failed	paused
1						
~						

Fig. 13.4 TableFile analysis

Preliminaries: With the help of the frame object, you can access all objects in the frame by an index (method: `<path>.node(<integer>)`). The method `<path>.numNodes` returns the number of objects within the frame. Finally, you can query the class of the objects with `<path>.class` (e.g. `.MaterialFlow.SingleProc`).

Add an `endSim` method to the frame. The method iterates through all objects in the frame. For objects of class `SingleProc`, the method inserts a row into the table "analysis" and writes its name and statistical values into it. Finally, the method exports the contents of the `TableFile` to an Excel file: `simulation_analysis.xls`.

With `getCurrentDirectory`, you can read the location of the Plant Simulation model.

```
--writes statistical data for all SingleProc objects
is
  i:integer;
  obj:object;
do
  analysis.delete;
  for i:=1 to current.numNodes loop
    obj:=current.node(i);
    if obj.class = .MaterialFlow.SingleProc then
      analysis.writeRow(0,
        analysis.YDim+1,
        obj.name,obj.statWorkingPortion,
        obj.statWaitingPortion,
        obj.statBlockingPortion,
        obj.statFailPortion,
        obj.statPausingPortion);
    end;
  next;
--write excel file
analysis.writeExcelFile(getCurrentDirectory+
  "\simulation_analysis.xls");
end;
```

You can easily extend this method by adding other classes. Set a simulation end of two days, and let the simulation run up to this point. Search for the Excel file (Fig. 13.5).

	A	B	C	D	E	F	
1	working	waiting	blocked	failed	paused		
2	<b>M1</b>	0,24513889	0,00131944	0,27460729	0,47893438	0	
3	<b>M2</b>	0,20055556	0,00294493	0,22738226	0,56911725	0	
4	<b>M3</b>	0,28400025	0,00311534	0,30084477	0,41203964	0	
5	<b>M4</b>	0,28361111	0,00340278	0,27649896	0,43648715	0	

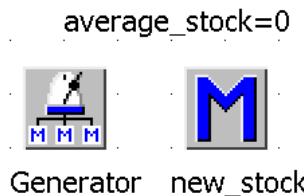
**Fig. 13.5** Excel export

### 13.2.2 Determining Average Values

Often, average values need to be calculated within a simulation. A typical example is the calculation of average stock. For an average calculation, you need a series of values and the number of the values (arithmetic average). Within the simulation, you can choose another approach. A generator calls a method every hour that determines the number of parts within the frame. The method calculates a new average based on the old average, the number of hours, and the new stock. You can easily find out the number of MUs in the frame with `<path>.NumMu` (Keep in mind that the method `numMU` also counts containers and transporters). A further possibility is the use of a TimeSequence in connection with the method `<timeSequence>.meanValue`.

#### Example: Statistics continuation

Add a variable `average_stock` (real) to the frame. Insert a generator into the frame and a method `new_stock`. The generator calls `new_stock` once every hour starting after one hour (Fig. 13.6).



**Fig. 13.6** Average stock

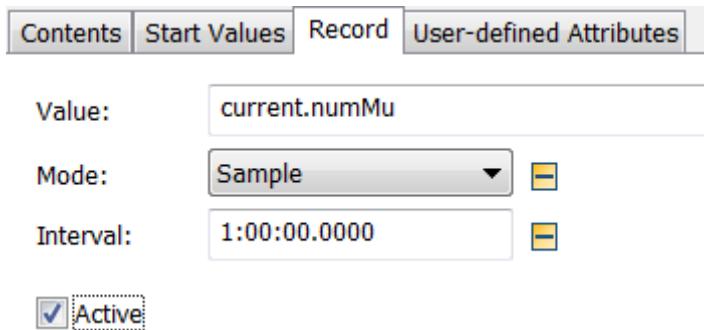
To calculate continuously a new average stock, you need programming like this:

```
is
  hours:integer;
do
  hours:=time_to_num(eventController.simTime)/3600;
  average_stock:=
    (average_stock*(hours-1)+current.numMu)/hours;
  record_failures;
end;
```

#### Average, variant TimeSequence

You can record the inventory at a regular interval. At the end of the simulation, you can calculate the average value from the TimeSequence. To do this, add a TimeSequence into your frame and a global variable `average_stock1` (real).

Format the second column of the TimeSequence as an integer. Ensure settings in the TimeSequence as in Fig. 13.7.



**Fig. 13.7** TimeSequence

In the endSim method, add the following command to calculate the average stock based on the TimeSequence:

```
average_Stock1:=TimeSequence.meanValue({2,1}..{2,*});
```

You must delete the content of the TimeSequence before each simulation run, because the TimeSequence is not deleted by the EventController automatically. The command for this is:

```
TimeSequence.delete
```

Note: For calculating the average stock, you can also use the attribute statRelativeOccupation in connection with capacity—e.g. avStock:=buffer.statRelativeOccupation\*buffer.capacity;

### 13.2.3 Record Values

For the evaluation and optimization of the simulation, you often need the progression of values over time. You can accomplish this in two ways:

- Record values with TimeSequence objects (a separate TimeSequence for each value)
- Record values with a table and analysis of the table

#### Example: Statistics continuation

In the example above, you are to show the distribution of the failures of the individual machines (hourly). If a failure has occurred, the value 1 should be entered; if the object is not failed at the moment, the value 0 should be entered. Insert the table failure\_machines into the frame and format it like in Fig. 13.8.

	time 1	integer 2	integer 3	integer 4	integer 5
string	time	M1_failure	M2_failure	M3_failure	M4_failure
1					

**Fig. 13.8** Table failure\_machines

Place a method record\_failures in the frame. A generator has to call this method every hour. Method record\_failures:

```

is
  i:integer; --next entry
do
  i:=failure_machines.YDim+1;
  --call once per hour
  failure_machines[1,i]:=eventController.simTime;
  -- set values depending of attribute failed
  if M1.failed then
    failure_machines["M1_failure",i]:=1;
  else
    failure_machines["M1_failure",i]:=0;
  end;
  if M2.failed then
    failure_machines["M2_failure",i]:=1;
  else
    failure_machines["M2_failure",i]:=0;
  end;
  -- and so on
end;

```

Similarly, you can record any other value.

### 13.2.4 Calculation of the Number of Jobs Executed Per Hour (JPH)

It is often necessary to determine the peaks of the workload, if it is not evenly distributed over time. A basis for the identification of such peaks could be the work or tasks executed per hour (jobs per hour, JPH). The jobs per hour can be determined relatively easily with the help of the attribute statNumIn.

#### Example: Jobs per hour

A line is fed via two stations. One station operates in two shifts and the other operates only in one shift. The hourly load of the line is to be recorded and evaluated. Create the frame from Fig. 13.9.

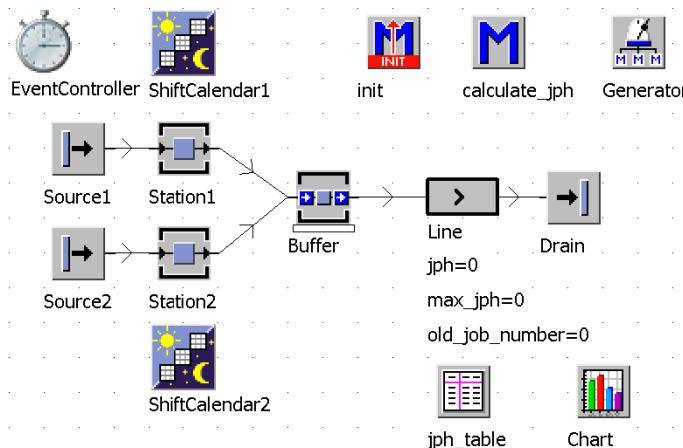


Fig. 13.9 Example frame

Ensure the following settings: Station1: processing time uniformly distributed between 15 and 60 seconds; ShiftCalendar1: one shift (default pauses); Station2: processing time uniformly distributed from 30 to 60 seconds; ShiftCalendar2 with two shifts (default pauses); buffer: 20 places, no processing time; Line: capacity 1, time 20 seconds. The generator calls calculate\_jph with an interval of one hour. The global\_variables jph, max\_jph and old\_job\_number each have the data type integer and a start value of zero. The first column of the table jph\_table has the data type time. The second column is of data type integer. The method init deletes all entries of the table jph\_table:

```
is
do
  jph_table.delete;
end;
```

To determine the JPH you need to record at the beginning of each hour (at the end of the preceding) the number of MUs that have entered the line. At the end of the hour you can compare the current number of MUs (statNumIn) with the number at the start of the hour (old\_job\_number) and calculate the JPH. If the current JPH is greater than the last saved maximum JPH, set a new value for the maximum JPH (max\_jph). For evaluating the curve of the JPH, save the current value and the corresponding time in the table jph\_table. The results are presented in the following programming of the method calculate\_jph:

```
is
do
  --calculate jph
  jph:=line.statNumIn-old_job_number;
  --save old job-number
  old_job_number:=line.statNumIn;
```

```
--max_jph ?
if max_jph < jph then
  --new max_jph
  max_jph:=jph;
end;
if jph > 0 then
  --new entry in jph table
  jph_table.writeRow(1,jph_table.yDim+1,
    eventController.simTime,jph);
end;
end;
```

### 13.2.5 Data Collected by the Drain

The drain collects detailed statistics about the destroyed parts. Open the drain, and select the tab Type Statistics (Fig. 13.10).

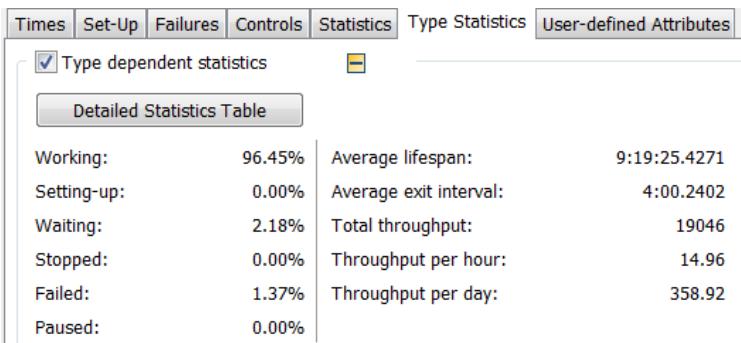


Fig. 13.10 Drain Type Statistics

Click the button Detailed Statistics Table to receive further information. The drain provides a number of methods for accessing statistics; Table 13.2 shows a small selection:

Table 13.3 Statistical methods of the drain

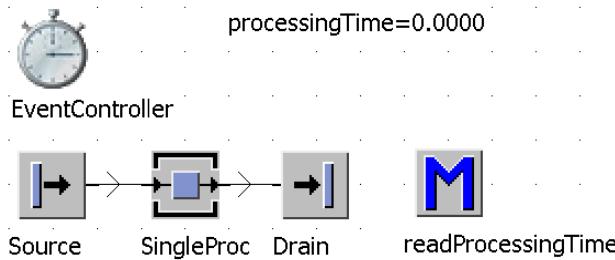
Method	Description
<path>.typeStatistics(<table>)	Copies the type statistics table in the specified table
<path>.typeStatisticsCumulated (<table>)	Analogous
<path>.statThroughputPerHour	Returns the throughput per hour (real)
<path>.ThroughputPerDay	Returns the throughput per day
<path>.statAvgLifeSpan	Returns the average dwell time
<path>.statAvgExitInterval	Returns the average exit interval (real average cycle-time)

### 13.2.6 Statistical Values of the Global Variable

You can use the variable block to collect statistical data. The block evaluates the frequency of occurrence of certain values. The statistics can be displayed as a histogram or table. The statistical recording of the variable block only works for the data type string and integer.

#### Example: Statistics function of the block variable

Create a simple frame like in Fig. 13.11.



**Fig. 13.11** Example frame

The source produces one part every minute. The SingleProc has a randomly distributed processing time. Enter the details of Fig. 13.12 in the processing time of the SingleProc.

Processing time:

**Fig. 13.12** Processing time lognorm

The variable `processingTime` has the data type string. The method `readProcessingTime` is the entrance control of the `SingleProc`. In the method, you read the current processing time of the single station, round it off and assign it as a text to the variable `processingTime`:

```
is
do
  processingTime:=
    to_str(round(SingleProc.procTime));
end;
```

Activate the statistics of the variable (check Active in the tab Statistics) and start the simulation. The variable block is now collecting statistical values (Fig. 13.13).

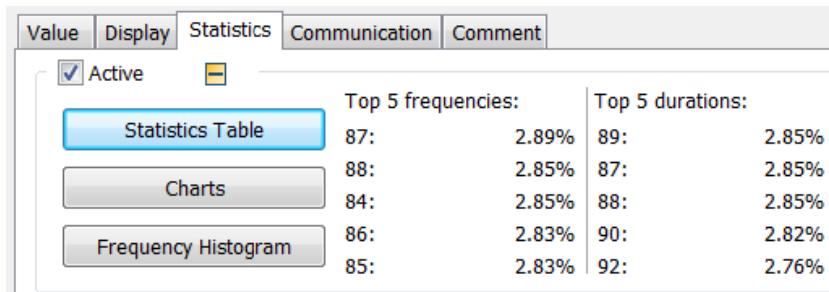


Fig. 13.13 Statistics of the global variable

### 13.2.7 Statistics of the Transporter

The transporter collects a large amount of statistical data. Especially in the simulation of transporters or workers (with the help of transporters), it is sometimes important to break down the simulation time further. For an automatic, battery-driven transport system, for example, the following statistical time components are required to assess the utilization:

- driving times
- loading and unloading times
- failure times
- battery loading times
- blocking times
- waiting times (without order)
- pausing times

There is no statistical value for loading and unloading in Plant Simulation. You can record a value by yourself, when you stop the transporter for this purpose. Plant Simulation provides statistical information for the transporter depending on the type of block on which the transporter is staying during the simulation time:

- statTransp...: Stay on a block of type transport (e.g. line, track)
- statProd...: Stay on a block of type production (e.g. SingleProc)
- statStore...: Stay on a block of type storage (e.g. buffer)

The category of the blocks is given in the attribute `resourceType`. This results in a very large number of possible statistical values for the transporter (Table 13.4). To see which kinds of statistical values Plant Simulation collects for the transporter, you can view the attributes and methods of the transporter after a full simulation run.

**Table 13.4** Statistics of the transporter (selection)

Method/Attribute	Description
<path>.statBatChargePortion	Percentage of battery charge time of the total simulation time
<path>.statTransportTimePortion	The statTransportTimePortion consists of statTranspWorkingPortion, statTranspWaitingPortion...; a portion of the simulation time in which the transporter was located on a track or a line (driving, stopped)
<path>.statTranspWorkingPortion	Driving on a track/line
<path>.statTranspWaitingPortion	Stopped on a track/line
<path>.statStoreWaitingPortion	Waiting on a buffer/store block
<path>.statTspFailPortion	Failed portion of the transporter
<path>.statTspPausingPortion	Paused portion of the transporter
<path>.statTspBlockingPortion	Blocked portion of the transporter

Further statistical data can be recorded during the simulation (e.g. waiting with/without MU).

### Example: Statistics of the transporter

You are requested to create a detailed report for a transporter. The basis for the example should be the transporter battery example. You can download it from [www.bangsow.de](http://www.bangsow.de). Change the following in the model:

1. Assign to the transporter in the class library an availability of 95 per cent with an MTTR of one minute.
2. 10 seconds of loading and unloading are taken into account. This should be considered by pausing the vehicle in the following example. Change the method `track.sensorControl` (path—user-defined attributes—`sensorControl`). Enter the following line in each case after the move command for sensors 2 and 3: `@.startPause(10);`
3. The waiting time of the loaded transporter should be shown separately. Add in the transporter in the class library a user-defined attribute (`statWaitingOccupied`; data type time). Add in the sensor control a variable declaration: `oldTime: time`. Change the sensor control method for sensor 3, as follows:

```
elseif sensorID = 3 then
  @.stopped:=true;
  oldTime:=ereignisverwalter.simTime;
  waituntil SingleProc.empty prio 1;
  @.statWaitingOccupied:-
    @.statWaitingOccupied+ereignisverwalter.simTime-
    oldTime;
  @.cont.move(singleProc);
```

```
@.startPause(10);
@.stopped:=false;
```

If the occupied transporter has to wait before unloading, the time is stored in the user-defined attribute.

4. Insert a method (endSim) and a table into the model. Format the table like in Fig. 13.14.

	string 0	real 1	real 2	real 3	real 4	real 5	real 6
string		Drive	Load/Unload	Charging battery	Failure	Wait occupied	Starved
1	Transporter1						
2	Transporter2						

**Fig. 13.14** Transporter statistics table

5. To be able to access the transporters at the end of the simulation easily, we require object references. This can be created in the easiest way when the transporters are created. To do this, add two global variables in the frame (Transporter1 and Transporter2, each data type object) and change the init method as follows:

```
is
do
  transporter1:=.MUs.transporter.create(track,8);
  transporter2:=.MUs.transporter.create(garage);
end;
```

The endSim method and the statistical data should be read from the transporters and written to the table. Driving time corresponds to the value of the attribute statTransportTimePortion, loading and unloading corresponds to statTspPausingPortion, battery charging time is statBatChargePortion, failure portion is found in statTspFailPortion, the proportion of waiting is equal to the attribute statWaitingOccupied in relation to the simulation time, the wait portion (empty) is the remainder to 100 per cent (or 1). The statistical analysis will result in a sum of one (100 per cent). Failures do not interrupt the battery charge times. Therefore, some failure events are getting lost. The same happens when failures and pauses occur at the same time in the simulation. Therefore, the recorded statistical values rarely correspond to the set values. Thus, the time allocation adds up to 100 per cent. You should calculate one statistical value as the remainder to 100 per cent. The endSim method should look like this:

```
is
do
  -- statistical values of transporter 1
  statisticsTable.writeRow(1,1,
    transporter1.statTransportTimePortion,
```

```

transporter1.statTspPausingPortion,
transporter1.statBatChargePortion,
transporter1.statTspFailPortion,
time_to_num(transporter1.statWaitingOccupied) /
time_to_num(eventcontroller.simTime),
-- diff to 1
1-transporter1.statTransportTimePortion-
transporter1.statTspPausingPortion-
transporter1.statBatChargePortion-
transporter1.statTspFailPortion-
time_to_num(transporter1.statWaitingOccupied) /
time_to_num(eventcontroller.simTime));
-- analogous for transporter2
end;

```

The table can then be relatively easily represented as a diagram (Fig. 13.15).



**Fig. 13.15** Diagram transporter statistics

### Idle time

Often, the time portion must be shown for a transporter (worker), when it is waiting empty and without a job (idle time). In order to separate the idle time of loading, unloading and any blockage times, you can, for example, use the statistics of a buffer. If the transporter always stays in a buffer when it has no order, then this time can be cleanly separated and evaluated from the rest of the waiting times. The relevant statistical value is `statStoreWaitingPortion`.

### Example: Transporter—Idle time

Create a frame according to Fig. 13.16.

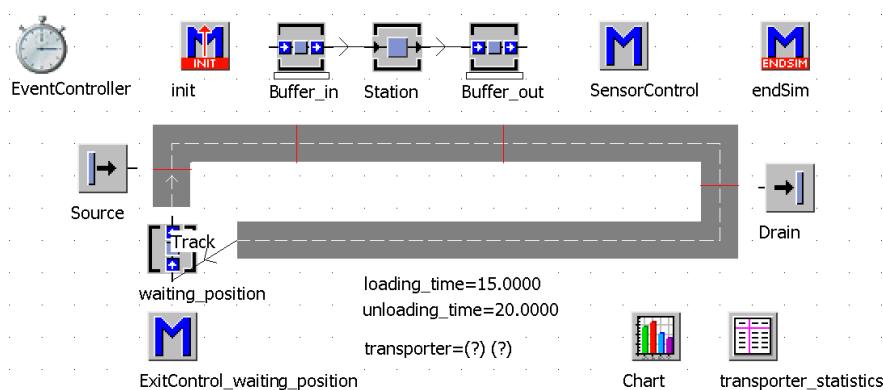


Fig. 13.16 Example frame

Create the following settings: Buffer\_in and Buffer\_out each one place, no processing time; station: processing time: 2:15 minutes; waiting\_position: one place, no processing time; exit control: method exitControl\_waiting\_position. Insert into the track four sensors, starting at the position of the source. Assign to each sensor the method SensorControl. Format the table transporter\_statistics like in Fig. 13.17.

	real 1	real 2	real 3
string	driving	loading/ unloading	idle
1			

Fig. 13.17 Statistics table

The init method creates a transporter on waiting\_position:

```
is
do
  transporter:=
    .MUS.Transporter.create(waiting_position);
end;
```

The SensorControl controls exactly one loop of the transporter:

- At the source, the transporter loads one part
- At the position of buffer\_in, one part is unloaded
- If the buffer\_out is occupied, the transport loads this part
- At the position of the drain, the transporter loads the part to the drain, if it is occupied

The loading and unloading times are modeled with the command wait (duration). This results in the following programming for the method SensorControl:

```
(SensorID : integer; Front : boolean)
is
do
  --one cycle
  if sensorID=1 then
    @.stopped:=true;
    wait(loading_time);
    source.cont.move(@);
    @.stopped:=false;
  elseif sensorID=2 then
    @.stopped:=true;
    wait(unloading_time);
    @.cont.move(Buffer_in);
    @.stopped:=false;
  elseif sensorID=3 then
    if Buffer_out.occupied then
      @.stopped:=true;
      wait(loading_time);
      buffer_out.cont.move(@);
      @.stopped:=false;
    end;
  elseif sensorID=4 then
    if @.occupied then
      @.stopped:=true;
      wait(unloading_time);
      @.cont.move(Drain);
      @.stopped:=false;
    end;
  end;
end;
```

At the end of the track, the transporter is moved to the waiting\_position. The exit control of waiting\_position waits until buffer\_in is empty. Then, it moves the transporter to the track. Thus, a new cycle for the transporter starts. The method ExitControl\_waiting\_position has the following content:

```
is
do
  waituntil buffer_in.empty prio 1;
  @.move(track);
end;
```

### Statistics

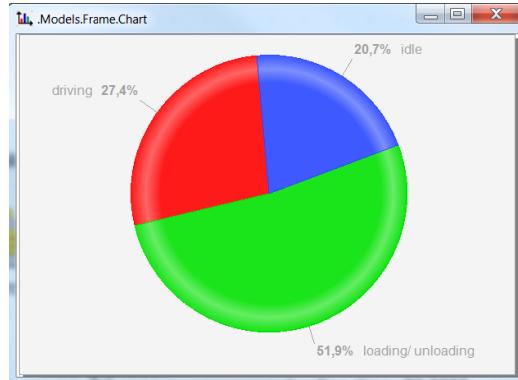
To evaluate the work of the transporter (excluding breaks and failures), three values are now available:

- the time in which the transporter was driving on the track (statTranspWorkingPortion)
- the time in which the transporter was stopped on the track for loading and unloading (statTranspWaitingPortion)
- the time in which the transporter has stayed on the buffer (without order) (statStoreWaitingPortion)

For evaluation (method endSim), these values are read and written into a table:

```
is
do
  transporter_statistics.writeRow(1,1,
    transporter.statTranspWorkingPortion,
    transporter.statTranspWaitingPortion,
    transporter.statStoreWaitingPortion);
end;
```

The values can be well-represented as a pie chart (Fig. 13.18).



**Fig. 13.18** Pie chart

### 13.3 User Interface Objects

Most statistical data can be understood more easily if they are presented graphically. For this purpose, Plant Simulation provides user interface objects.

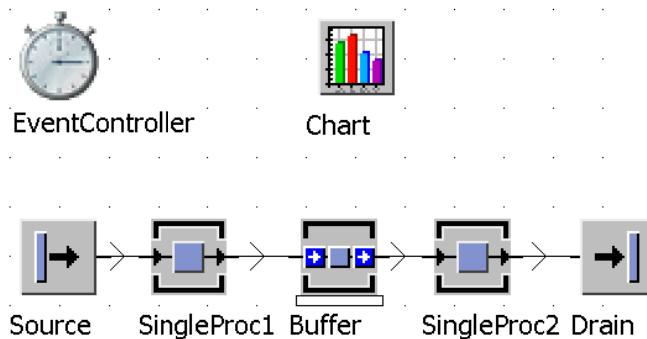
#### 13.3.1 The Chart Object

The object Chart represents data in Plant Simulation graphically. The data to be displayed can be located either in tables or you define input channels in the diagram that refer to specific values. In the watch mode, the graphic is automatically updated after each modification of a displayed value (the value must be observable).

In this way, you can visualize the dynamic behavior of certain values during the simulation.

### Example: Chart

In the following example, the development of stock in a buffer is to be presented graphically. Create a frame like in Fig. 13.19.



**Fig. 13.19** Example frame

Settings: SingleProc1, SingleProc2 processing time of one minute each; availability for SingleProc1 and SingleProc2 of 95 per cent; MTTR of 30 minutes; buffer capacity of 100, accumulating; 30 seconds processing time. There are two ways to present data in charts: from input channels or from Table-Files. An input channel means that the Chart object itself records and displays the data. You can access the recorded values using SimTalk.

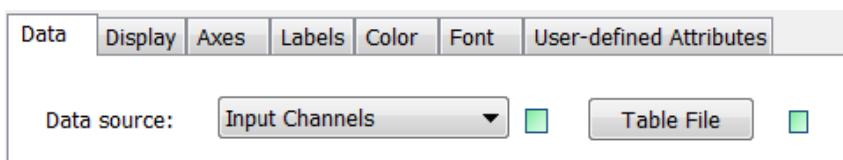
#### 13.3.1.1 Plotter

You can visualize in a diagram (plotter mode) one or more values and continually update this view. Plant Simulation plots these values in a chart window.

### Example: Chart continuation

You are to plot the stock in the PlaceBuffer.

1. Click the tab Data in the Chart and select Data source—Input Channels



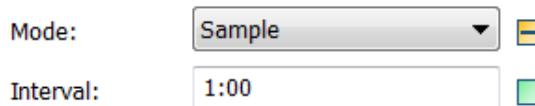
**Fig. 13.20** Chart tab Data

Clicking the button opens a table into which you can enter the name of the object, the path of the displayed value and comments. First, turn off inheritance of the table (+ Apply). Then click the button Table File. Enter the path and the attribute that is to be displayed (Buffer.numMU, in this example; see Fig. 13.21).

string	0	string
string		Buffer stock
1	Number	Buffer.numMU

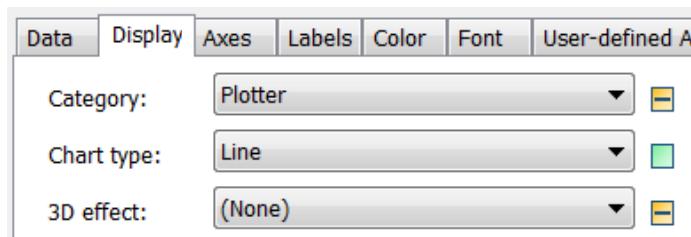
**Fig. 13.21** Input channel

You still need to select at which interval Plant Simulation is to update the chart on the tab Data. Watch mode updates after every change of the observable value, sample mode updates within the set interval, and plot mode updates the graph after each simulation event. In the example, the chart will be updated every minute, the setting for this is like in Fig. 13.22.



**Fig. 13.22** Mode sample

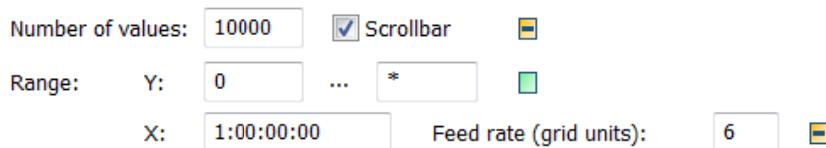
Select the category plotter and Chart type line on the tab Display (Fig. 13.23).



**Fig. 13.23** Display—Plotter

If you select the option Display in a frame, Plant Simulation shows the diagram (with its values) instead of the object's icon in the frame.

You then have to select some settings on the tab Axes. Enter the number of displayed values (e.g. 10,000). The scroll bar option is mostly useful for presentations using a plotter. In addition, you have to enter the size of the displayed time range in the plotter window in the box Range X; enter 1:00:00:00 (one day, Fig. 13.24).



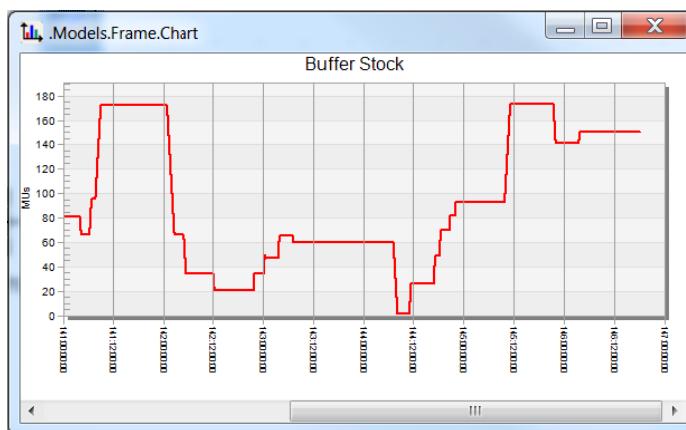
**Fig. 13.24** Settings axis

Add labels to the plotter on the tab Labels. Here, you can also specify whether a legend is displayed and how it is displayed in the chart window (Fig. 13.25).



**Fig. 13.25** Labels

You can select additional format settings on the tab Font. Clicking the button Show Chart opens the window of the plotter (Fig. 13.26).



**Fig. 13.26** Plotter

You can access the data that the plotter records and write it into a TableFile. Add a TableFile (analysis) and a method (saveData) to the frame. You can read the data of the chart object with the method:

```
<path>.putValuesIntoTable(<table>)
```

First, turn off inheritance for the TableFile. Program the method saveData:

```
is
do
  -- delete previous values
  analysis.delete;
  -- write the chart data into the table analysis
  chart.putValuesIntoTable(analysis);
end;
```

Plant Simulation formats the table and inserts the data of the plotter.

### 13.3.1.2 Chart

Plant Simulation provides a large number of different chart types to display values.

#### Example: Chart continuation

You are to show the composition of the statistics collection period of the objects SingleProc1 and SingleProc2. You need to write the values into a TableFile and show the values of the table in a chart. The values from Table 13.5 will be displayed:

**Table 13.5** Statistical values

Value	SimTalk Attribute
Waiting time	<path>.statWaitingPortion
Working time	<path>.statWorkingPortion
Blocked time	<path>.statBlockingPortion
Failed time	<path>.statFailPortion
Paused time	<path>.statPausingPortion

Insert a TableFile (chart\_values) into the frame. Format it like in Fig. 13.27.

	string 0	real 1	real 2	real 3	real 4	real 5
string		waiting	working	blocked	failed	paused
1	SingleProc1					
2	SingleProc2					

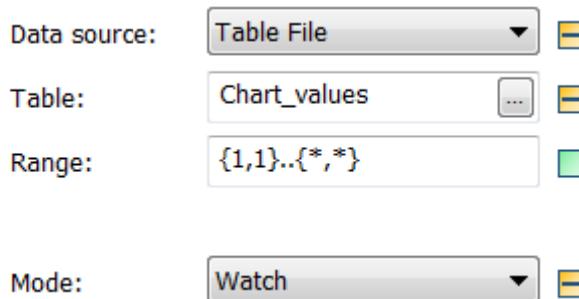
**Fig. 13.27** Table chart\_values

At the end of the simulation run, a method (endSim) will write the statistics values of the two SingleProc into the TableFile chart\_values. Program the method endSim:

```
is
do
  -- values of SingleProc1
  chart_values.writeRow(1,1,
    SingleProc1.statWaitingPortion,
    SingleProc1.statWorkingPortion,
    SingleProc1.statBlockingPortion,
    SingleProc1.statFailPortion,
    SingleProc1.statPausingPortion);
  -- values of SingleProc2, next row
  chart_values.writeRow(1,2,
    SingleProc2.statWaitingPortion,
    SingleProc2.statWorkingPortion,
    SingleProc2.statBlockingPortion,
    SingleProc2.statFailPortion,
    SingleProc2.statPausingPortion);
end;
```

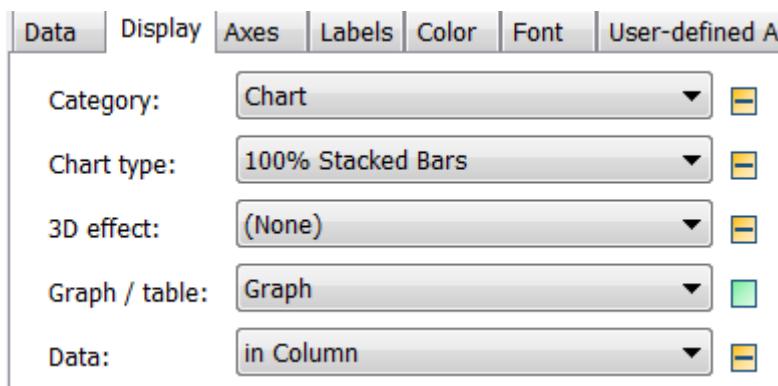
### Creating Charts

Insert a Chart object into the frame. Open the chart and select the Data source—TableFile on the tab Data. Enter the name of your table, Chart\_values, into the field Table (Fig. 13.28).



**Fig. 13.28** Setting data

Select Category > Chart on the tab Display. The chart-type “100% stacked bars” is suited for displaying the portions of the statistics collection period. You need to set Data—In Column (structures of the table chart\_values—the data are spread across several columns; see Fig. 13.29).



**Fig. 13.29** Display settings

Label your chart. Show the legend on the tab Labels. To avoid misunderstandings, you must customize the colors in the chart so that they match the colors of the status LEDs (in any case, failures should be shown in red, pauses in blue, and blockages in orange). You can set the order of colors on the tab Color. Double-click a color to change it, and then select a new color. The order of colors in the example above must be yellow, green, orange, red, and blue (Fig. 13.30).

Color	Opacity	Line Style	Line Weight	Marker Type
1	255	—	Thin	Solid Circle
2	255	—	Thin	Solid Square
3	255	—	Thin	Solid Diamond
4	255	—	Thin	Solid Triangl...
5	255	—	Thin	Solid Triangl...

**Fig. 13.30** Chart colors

Assign a ShiftCalendar to SingleProc1 and SingleProc2 (default settings). Reduce the Buffer to a capacity of 20. Set in the EventController an end of 100 days.

The chart will be shown by clicking the button Show chart. You can also show it by using the context menu of the chart object icon (Fig. 13.31).

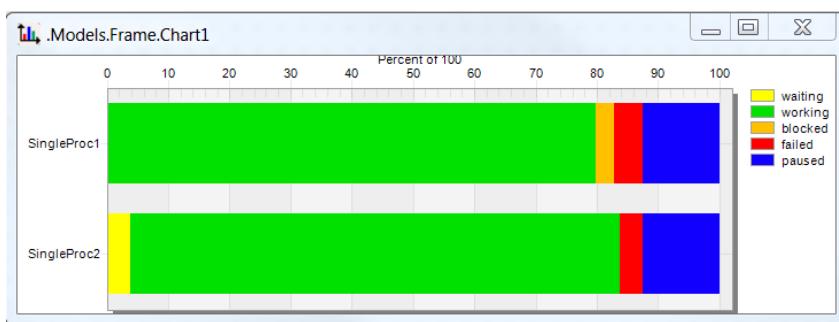


Fig. 13.31 Chart

### 13.3.1.3 Statistics Wizard

If you want to analyze the statistics collection period of all objects of a certain class, you can use the statistics wizard. For that, add a chart object to the frame. Right click on the chart icon in the frame. Select Statistics Wizard from the context menu (Fig. 13.32).

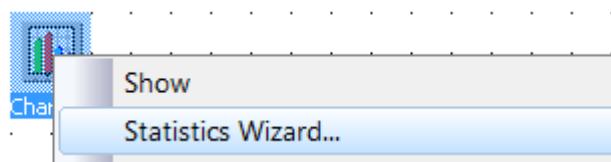


Fig. 13.32 Statistics Wizard

Select the objects whose statistics you want to show in the dialog of the statistics wizard. Leave SingleProc and Production checked (Fig. 13.33).

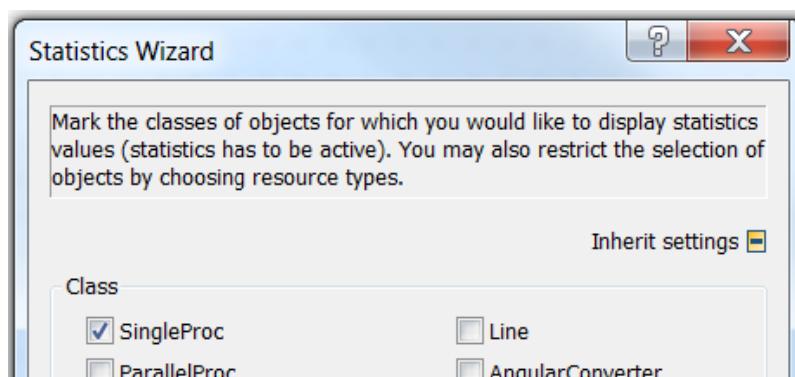


Fig. 13.33 Statistics Wizard

### 13.3.1.4 Histogram

Histograms show the frequency of certain values in relation to the simulation time. You need to display a histogram of a larger set of values.

#### Example: Chart continuation

You want to display the distribution of the occupancy of the Buffer (attribute numMU). Use the following steps:

Add a new chart to the frame. Select Data—Input Channels and enter Buffer.numMU in the table (column 1).

Select Category—Histogram and the Chart Type—Columns on the tab Display (Fig. 13.34).

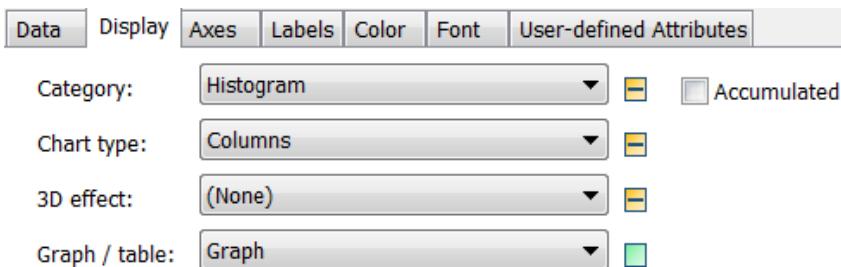


Fig. 13.34 Setting for histogram

Clicking Show Chart displays the histogram (Fig. 13.35).

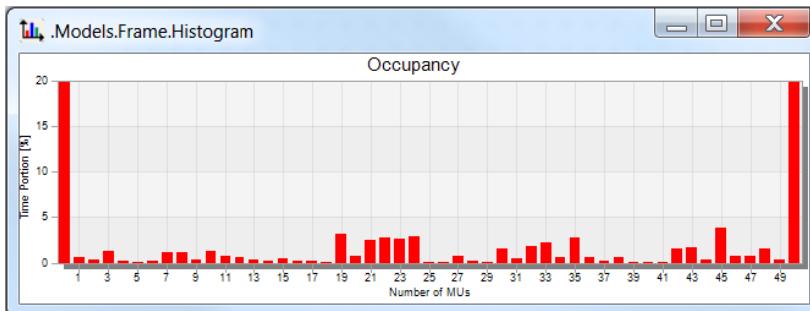


Fig. 13.35 Histogram

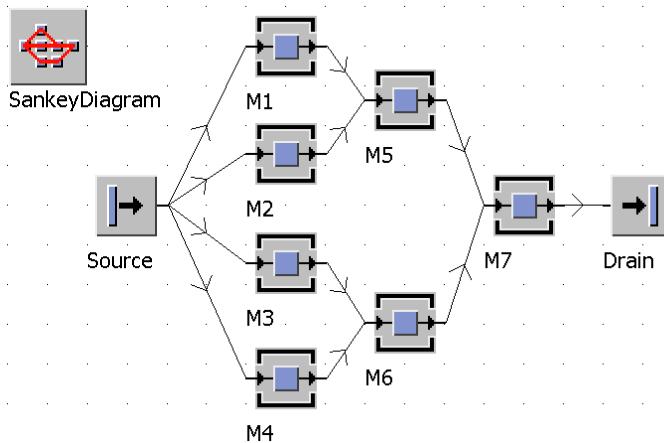
**Note:** You can copy charts in the dialog box of the chart object to the clipboard and then paste them into other programs (e.g. PowerPoint). Select Tools—Copy to Clipboard in the dialog of the Chart.

### 13.3.2 The Sankey Diagram

The Sankey diagram is used for visualizing the distribution of the material flow. For this, Plant Simulation uses lines with different widths. The Sankey diagram is located in the folder Tools, or on the toolbar Tools.

#### Example: Sankey diagram

Create a frame according to Fig. 13.36.



**Fig. 13.36** Example frame

Ensure the following settings:

**Table 13.6** Example settings

Machine	Processing time	Availability	MTTR
M1	1:00.0000	95%	2:00:00.0000
M2	1:00.0000	85%	2:00:00.0000
M3	1:00.0000	70%	2:00:00.0000
M4	1:00.0000	50%	2:00:00.0000
M5	1:00.0000	95%	2:00:00.0000
M6	1:00.0000	85%	2:00:00.0000
M7	50.0000	95%	2:00:00.0000

The source produces parts with an interval of one minute (blocking) while the exit strategy is cyclic blocking. Add a Sankey diagram to the frame. Open the Sankey diagram by double-clicking it. Click the button Open (MUs to be watched). Enter the MU class, which is to be observed, into the following table.

Drag the class Entity from the class library into the table. You can select some formatting options, such as color settings and the maximum width of the lines (Fig. 13.37).

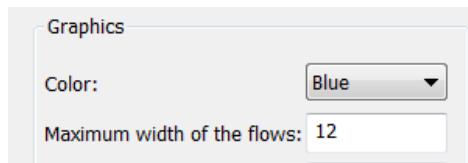


Fig. 13.37 Sankey diagram settings

Graphics in layer determines the z-position of the Sankey display. The smaller the number, the closer to the foreground a graphic is located. Complete your settings by clicking OK. Now, run the simulation for a while (50 days). Then right click on the object Sankey diagram in your frame. Select Display Sankey Diagram (Fig. 13.38). Delete Sankey Diagram deletes the Sankey streams.

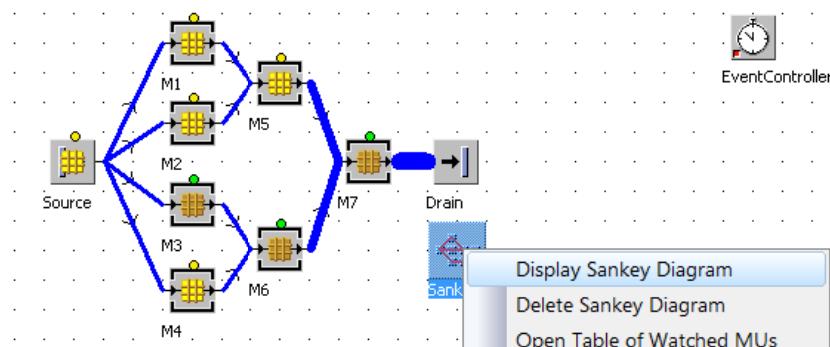


Fig. 13.38 Display Sankey Diagram

The thicker the Sankey streams between two stations, the more MUs have been transported on the connectors or methods between these stations. The exit strategy cycle of the source leads to the stations M1 to M4 receiving the same number of parts. If a machine fails, the source waits with the transfer process until the machine is operational again. M1 to M4 receive the same number of parts (Sankey lines have the same width). The output after 50 days is around 21,000 parts.

You are to simulate a second variant. Right click on the frame in the class library, and select Duplicate. Close the frame window and open the duplicate. Change this frame as in Fig. 13.39.

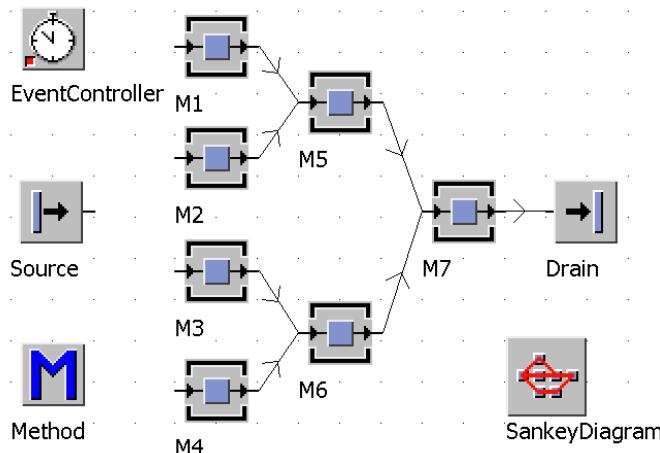


Fig. 13.39 Example frame

Program the method and assign it as the exit control (front) to the source. The source has to transfer the parts to the first available and operational machine:

```

is
do
  waituntil (m1.operational and m1.empty) or
  (m2.operational and m2.empty) or
  (m3.operational and m3.empty) or
  (m4.operational and m4.empty)  prio 1;
  if (m1.operational and m1.empty) then
    @.move(m1);
  elseif (m2.operational and m2.empty) then
    @.move(m2);
  elseif (m3.operational and m3.empty) then
    @.move(m3);
  elseif (m4.operational and m4.empty) then
    @.move(m4);
  end;
end;

```

Run this simulation for 50 days. The output is nearly 72,000 parts. The Sankey diagram now reflects the availability of the machines (the lower the availability, the fewer parts run across the machines, Fig. 13.40).

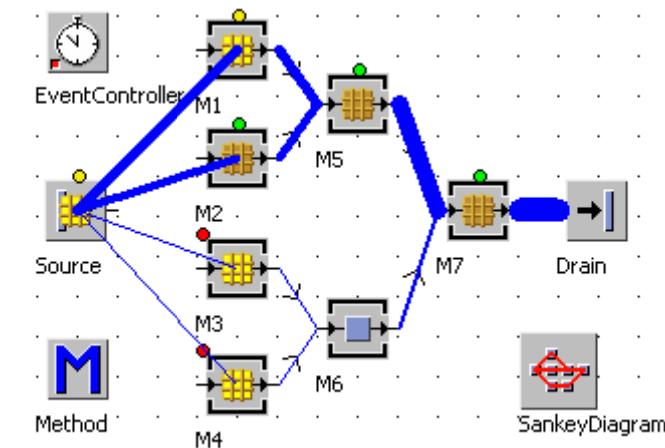


Fig. 13.40 Sankey diagram

### 13.3.3 The BottleneckAnalyzer

The BottleneckAnalyzer visualizes the default statistics for all selected objects. It is quite simple to use. First, make sure that sufficient space is available above the top object. Insert a BottleneckAnalyzer object into your frame (folder tools). Open the object BottleneckAnalyzer and click the tab Configure. Select object types for which you want to display statistics (Fig. 13.41).

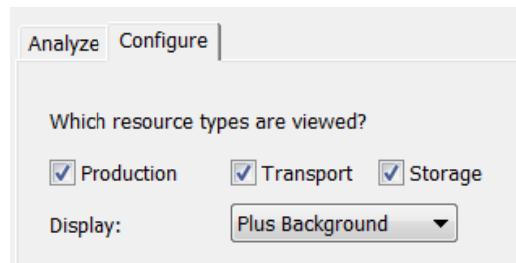


Fig. 13.41 BottleneckAnalyzer configuration

Click the button Analyze on the tab Analyze to create the statistics evaluation. The statistical data is displayed graphically in the frame (Fig. 13.42). You can also output the data after the analysis as a table. Click Ranking Table—Open. Once you have chosen a sorting option, the table is displayed.

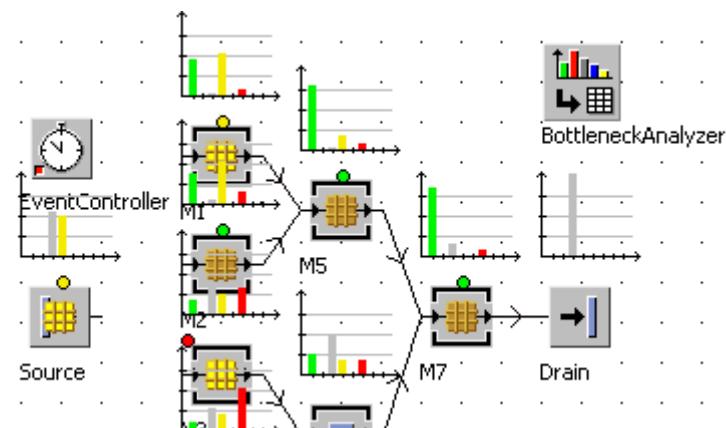


Fig. 13.42 BottleneckAnalyzer visualization

### 13.3.4 The Display

You can use the display object to show dynamic values (attributes, variables) during a simulation run. The values can be represented as a number or bar. The activated object periodically checks the value and updates the display (Sample mode) or after a corresponding change (Watch mode). As a bar or pie, the display shows numeric values in relation to the specified interval (between minimum and maximum).

#### Example: Display

Create a simple frame like in Fig. 13.43.

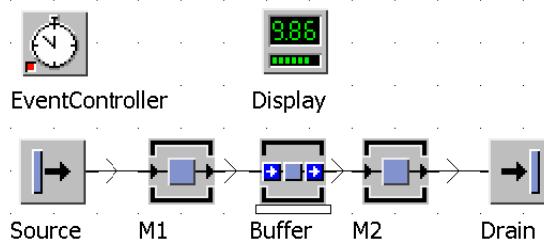
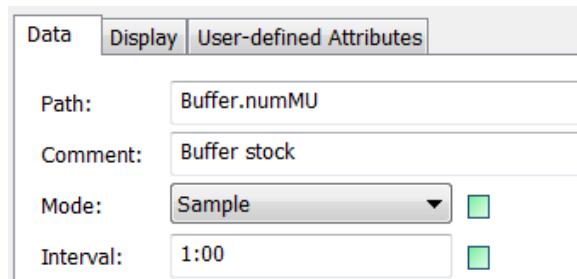


Fig. 13.43 Frame example

Settings: Source interval: 2:10 minutes; M1 processing time: 2:00 minutes, availability: 90 percent, one hour MTTR; M2: one minute processing time, 50 per cent availability, two hours MTTR; buffer capacity: 1,000, no processing time. The display should show the stock of the buffer. Open the display block by double-clicking. Enter `buffer.numMU` in the field path and "Buffer stock" in the field comment (Fig. 13.44).



**Fig. 13.44** Display data

Enable the display object by clicking on Active. If you run the simulation now, the buffer stock is displayed above the buffer.

The main settings of the display are:

- Path: Enter the path to the observed value (relative or absolute). You can enter global variables, attributes, and methods (invalid paths are marked).
- Comment: Enter a detailed description of the display that appears on the left-hand side of the value.
- Mode: Select Watch or Sample mode (with interval).

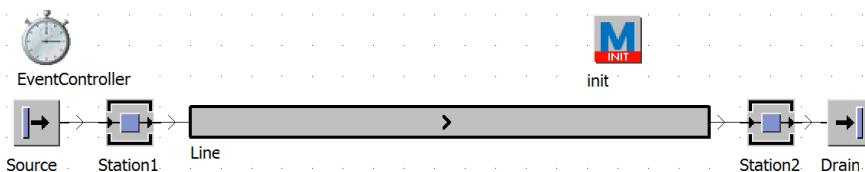
The value of the display can be shown as a bar or pie graph (numeric values), or as text. The bar/pie graph shows the ratio of the actual value to a given maximum value.

### 13.3.5 The Display Panel

You can equip the most blocks in Plant Simulation with a display panel. On this display panel, you can represent a large amount of information.

#### Example: Display Panel

In the following example, an accumulating conveyor line is used for decoupling of two machines. You should display in the model the occupied length, the number of MUs on the conveyor line and the total weight of all MUs located on the conveyor line. Create a frame as in Fig. 13.45.



**Fig. 13.45** Example frame

Create the following settings:

The station1 has a processing time of 1:30 minutes, the conveyor line is 17 meters long, speed 1 m/sec.; Station2 has a uniformly distributed processing time ranging from 45 seconds to 120 seconds, and an availability of 90 per cent with an MTTR of 10 minutes. Change the length of the entity in the class library to 50 cm. Insert into it a user-defined attribute (mu\_weight: weight). Set the weight to five kilograms. Add in the conveying line a user-defined attribute (occupation\_weight: weight). Equip the line with entrance control (rear) and exit control (rear).

### Calculate the Weight

The easiest way to calculate the total weight of the parts on the conveyor line is by increasing the weight on the line at the entrance of a part and to reduce it after a part has left the line. Therefore, the input control and output control (rear) are suitable. The entrance control of the line would look like this:

```
is
do
    ?.occupation_weight:=
        ?.occupation_weight+@.mu_weight;
end;
```

Exit control:

```
is
do
    ?.occupation_weight:=?.occupation_weight-
        @.mu_weight;
end;
```

The init method must set the attribute occupation\_weight to zero at the beginning of the experiment:

```
is
do
    line.occupation_weight:=0;
end;
```

The display panel can be activated via the context menu of the line—Edit display panel. A dialog is open, which allows you to enable and configure the display panel (Fig. 13.46).

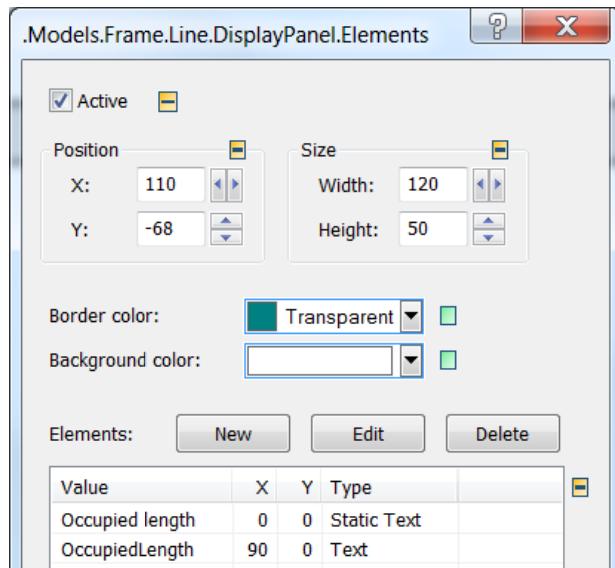


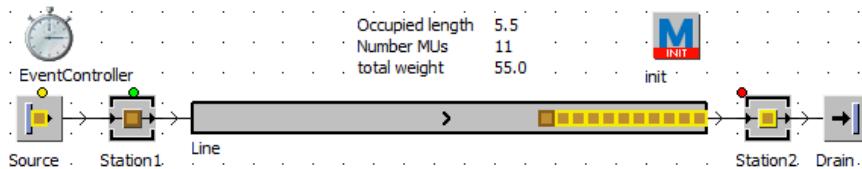
Fig. 13.46 Setup display panel

Enable the display panel by checking the box Active. In the area position you set the position of the display panel relative to the line (in pixels). The size refers to the entire display panel. The display panel can show static text (labels), text (values of attributes, results of methods), bars or LEDs (Boolean values). You can visualize all attributes of the object (including user-defined attributes). You must specify in the definition the elements to be displayed, the position within the display panel, the displayed value and the type of display. Define the contents of the display panel based on the table in Fig. 13.47.

	Value	X	Y	Type	Alignment
1	Occupied length	0	0	Static Text	Left
2	OccupiedLength	90	0	Text	Left
3	Number MUs	0	14	Static Text	Left
4	NumMU	90	14	Text	Left
5	total weight	0	28	Static Text	Left
6	occupation_weight	90	28	Text	Left

Fig. 13.47 Display panel data

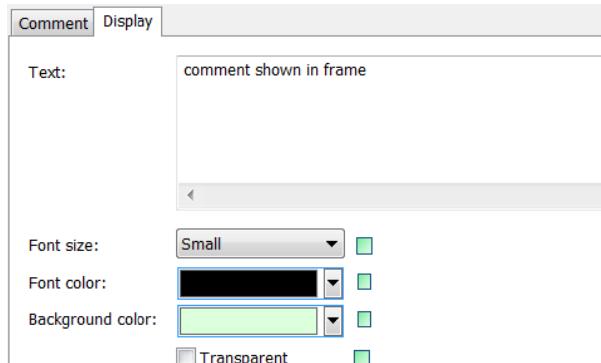
The current values are now displayed above the line (Fig. 13.48).



**Fig. 13.48** Display panel

### 13.3.6 The Comment

The comment has no active behavior during the simulation run and can be used for explanations and labeling. The comment block has two tabs: Comment and Display (Fig. 13.49).



**Fig. 13.49** Comment block

The text, which you enter here, will be shown in the frame. You can assign the text dynamically using the method `<path>.text:= <string>`. If Transparent is selected, the comment is shown within a box with the background color shining through. In the big text box on the tab Comment, you can save more text, which is visible only after opening the comment object or via the context menu of the comment—Open comment window (Fig. 13.50).

The comment text can be created in Rich Text Format (e.g. you create the text in Word and paste it into the comment via the clipboard). You can find formatting options in the context menu of the input field. You can access the contents of the comment with `<path>.cont.`

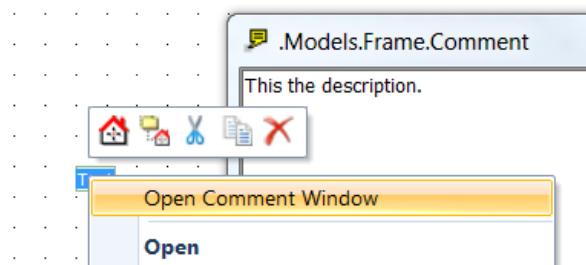


Fig. 13.50 Comment window

## 13.4 The Report

A report can present a large number of data. The report consists of header data and the report data, which you can arrange hierarchically. From Version 12 on, Plant Simulation provides a new report generator.

### 13.4.1 Automatic Resource Report (Statistics Report)

You can automatically create reports in Plant Simulation. Select the objects for which you want to create a report while holding down the Shift key. Then press the F6 key (Fig. 13.51).

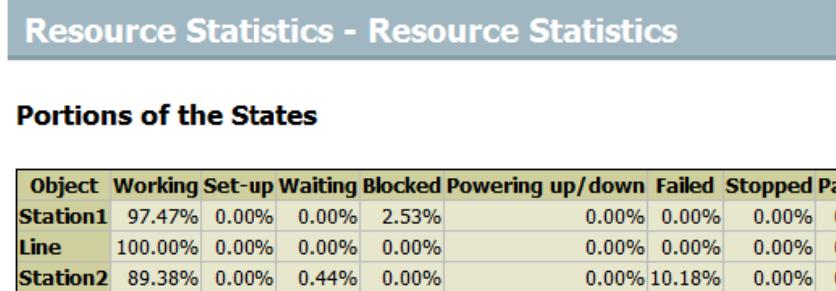


Fig. 13.51 Resource Statistics

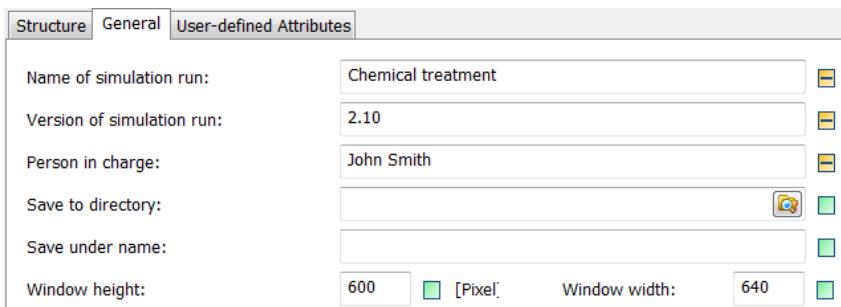
### 13.4.2 The Report Until V. 11

#### Report Header

The report header contains important information that is displayed in each page of the report, such as the name of the simulation, the name of the person responsible and version information. You can also display images in the report header.

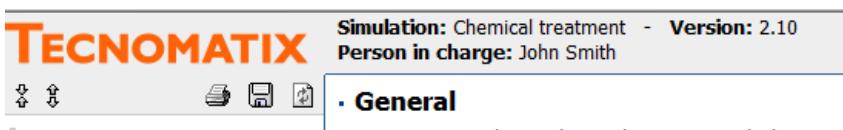
**Example: Report V. 11**

Use the example statistics for creating the report. Insert a report into the frame and open the report by double-clicking it. Insert general information about the simulation on the tab General (Fig. 13.52).



**Fig. 13.52** Report header settings

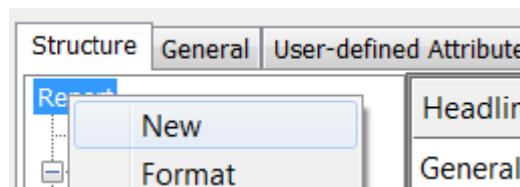
These data are shown later in the report header (Fig. 13.53).



**Fig. 13.53** Report header

**Report Data**

Define the structure of the report and the displayed information on the tab Structure. In the left pane, you can add new pages using the context menu of each (folder) object. You must first turn off inheritance. Right click on Report and select New from the context menu (Fig. 13.54).

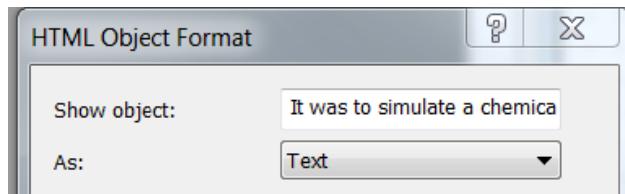


**Fig. 13.54** Report—New element

Rename the first sheet of the report as General. Right click on the entry. Select Rename from the context menu. Then, you can overwrite the name of the page. The data of each page will be structured using headings. Each page is separated into three columns. Each column can either contain an icon, text, or an object call.

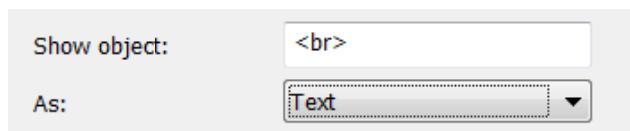
The first page includes a brief description of the simulation and a screenshot of the frame. Type "General" in the box Headline, then double-click in the box next to it (column 1). You can place text, icons, or method calls in the report. Select the format Text for the first field (first row, first column).

Type the following text into the box Show object: "It was to simulate a chemical treatment, which is supplied by four machines. Through a complex work process, the machines have an availability of only 50 per cent" (Fig. 13.55).



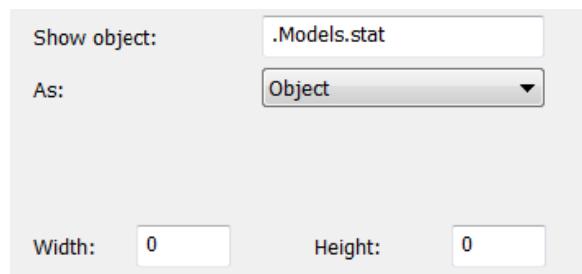
**Fig. 13.55** Report content

The next line is to remain free. You can insert HTML tags as text; the HTML command for a blank line is `<br>`. The report consists of HTML pages. Embedded HTML instructions accordingly modify the appearance of the report. Type the content from Fig. 13.56 into the second row, first column.



**Fig. 13.56** Report content

A screenshot of the frame will be shown below. Pressing Enter in the last row, last column of the table on the tab structure creates a new row. Enter the content like in Fig. 13.57 into the first column of the third row.



**Fig. 13.57** Report model screenshot

Enter the address of your frame in the class library. Set width and height to zero. Plant Simulation then determines the width and height of the image. The structure of the report page should look as in Fig. 13.58.

Headline	First column	Second col
General	It was to simulate a chem...	
Frame	.Models.stat	

Fig. 13.58 Report structure

You can view the report by clicking the button Show Report (Fig. 13.59).

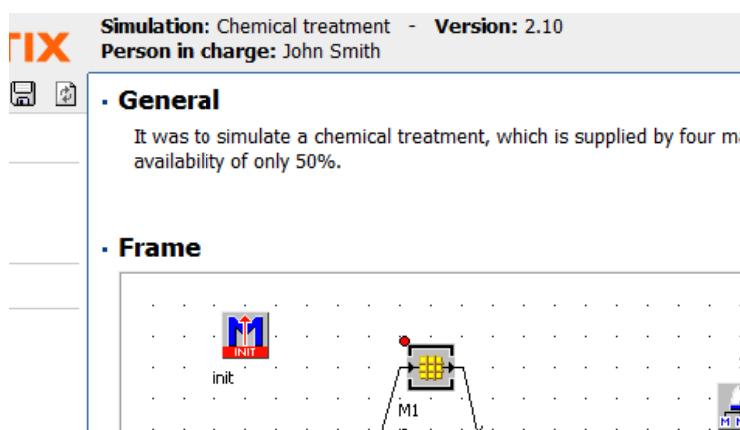


Fig. 13.59 Report

### Texts in Reports

You can enter text directly into the report (see above) or use the comment object for inputting text and outputting the contents of the comment block in the report.

Example: Insert a comment object into the frame, and type text like in Fig. 13.60 into the tab Comment.

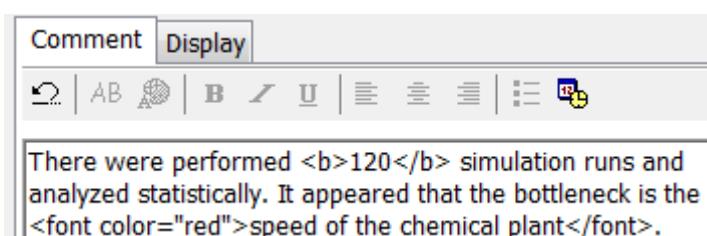


Fig. 13.60 Report text with HTML commands

Disable the option “Save the content in rich-text format” in the comment object. Add a page to the report and name it “Evaluations.” Then right click on the new page and add another page with New (General, Fig. 13.61).

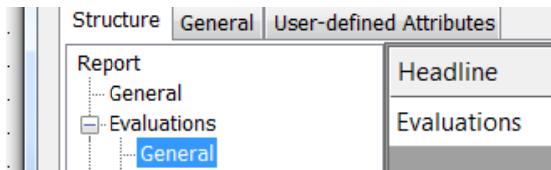


Fig. 13.61 Report data

The title of the page should be "Evaluations." In the first column of the first row, enter the reference to the content of the comment block: comment.cont (Fig. 13.62).



Fig. 13.62 Report data

The content of the comment is shown with the included HTML formatting in the report (Fig. 13.63).

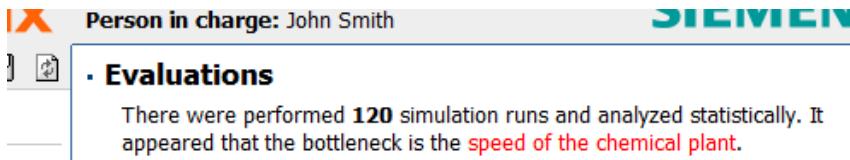


Fig. 13.63 HTML-formatted text

You can use the following HTML commands for formatting your text (selection).

**Table 13.7** HTML tags

HTML tag	Description
<H1> Headline 1 </H1>	Headline outline levels 1 to 6
<BIG>Text </BIG>	Rel. enlarged text
<SMALL>Text </SMALL>	Smaller text
<FONT SIZE=7>Text </FONT>	Font size (1 [very small] to 7 [very large])
<FONT COLOR="red">Text </FONT>	Font color as color or hexadecimal
<FONT FACE="Arial, Tahoma"> Text</FONT>	Font face as list separated by commas
<B>Text</B>	<b>Bold</b>
<I>Text</I>	<i>Italic</i>
<U>Text</U>	Underlined text
<S>Text</S>	Strike through
<SUB>Text</SUB>	Subscript
<SUP>Text</SUP>	Superscript

Combination: <B><I><U> bold, italic, underlined </U></I></B>

### Show Objects in Reports

You can access attribute values in reports and display objects. When you display objects in reports, Plant Simulation creates an image of the object (e.g. graph, frame), or displays values of default attributes. You can type in a complete SimTalk call into the field Show Object.

Example: Insert the page Statistics into the report. Here, you are to display the main statistical data of the machines. The headline is the name of the machine. The first column should display the names of the values in the second column and in the third column the units of the values. At the end of the page, a chart with the statistical data is to be displayed. To display the value of statWorkingPortion in the second column, enter the following settings:

- Show object: round(M1.statWorkingPortion\*100,2)
- Show as Object

Note: The default setting in the report is Show as Object. If you want to display text, you must switch to Show Object as Text; otherwise, you get an error when calling the report. The settings for displaying statistical data for machine M1 would look like Fig. 13.64.

H...	First column	Second column	T
M1	Portion working	round(M1.statWorkingPortion*100,2)	%
	Portion waiting	round(M1.statWaitingPortion*100,2)	%
	Portion fail	round(M1.statFailPortion*100,2)	%
	Portion blocked	round(M1.statBlockingPortion*100,2)	%

Fig. 13.64 Report data

Report (Fig. 13.65).

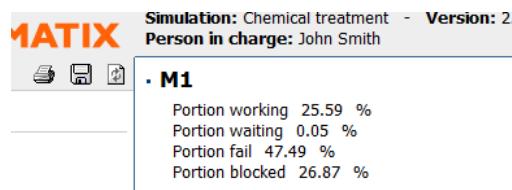


Fig. 13.65 Report

Charts and tables are inserted into the report via a simple object call. When inserting a chart, you must specify a size for displaying it. Example: The chart object has the name “Chart” and the necessary setting in the report resembles Fig. 13.66.

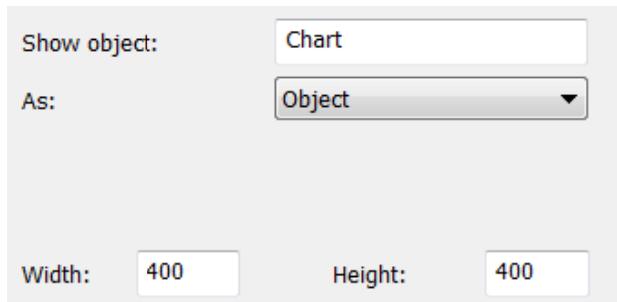


Fig. 13.66 Report data

To display methods, we have to use a little trick. You can access the text of the method as follows:

```
ref(<path>).program
```

The attribute program returns the entire text of the method, including the control characters. The control characters are normally ignored in the HTML display; hence, there will be a presentation without line breaks and tabs. With the HTML statement `<pre>text</pre>`, you can force the report to display line breaks and

control characters. If you want to display a method in the report, use a setting like in Fig. 13.67 (display as object).

Headline	First column
Method	"<pre>" + ref(method).program + "</pre>"

Fig. 13.67 Report data

### Show Images in Reports

You can also display images in reports. The images must be created as icons of the report (Context menu—Edit icons—Icon—New; File—Open). You must specify the icon number when inserting it into the report—e.g. the image is saved as an icon in the object Report (icon No. 16; see Fig. 13.68).

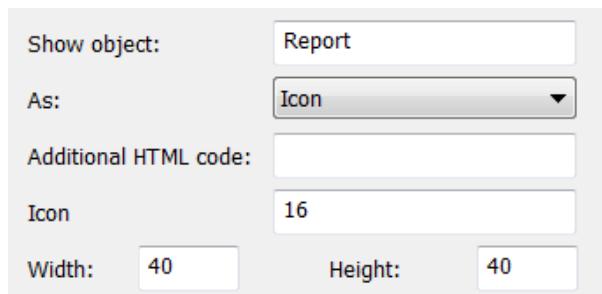


Fig. 13.68 Images in reports

### 13.4.3 The HTML Report (V. 12)

With Version 12, a new report in Plant Simulation was introduced (HTMLReport). The report will now be edited in a text editor such as with a scripting language (Fig. 13.69).

```
[!self, Header, *]
# General Information
## Overview
* Model file: [=modelFile]
* Simulation root: [EventController.Location]
* Simulation time: [EventController.SimTime]
```

Fig. 13.69 HTMLReport editor

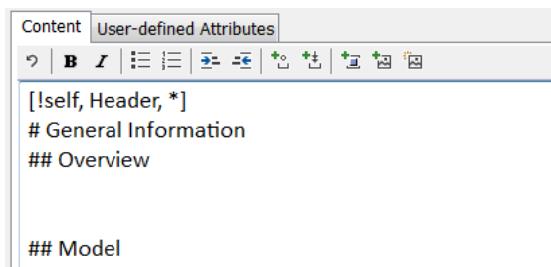
Different tags are used to control the structure and the format of the report. In addition, HTML elements can also be defined. The following tags are used in HTMLReport (selection, Table 13.8):

**Table 13.8** HTMLReport tags

Tag	Description
#	First-level heading; at the same time, menu item in the navigation
##	Second-level heading; at the same time, menu item in the navigation
*	Bullet-chart item
1.	Numbered item
**...**	Bold
*...*	Italic
><[object,*]	Screenshot of an object, the second parameter specifies the size (see Help)
[objectClass*]	Standard statistics of an object; you need to specify the object class
[=expression]	Result of an expression (e.g. calculation, method, attribute)
[!object,"IconName"]	Icon of an object (image)
--	Comment

### Example: HTMLReport

Use the example statistics for creating the report. Insert a report into the frame and open the report by double-clicking it. You will find plenty of standard content in the report. First, delete some rows until you see the content as shown in Fig. 13.70.



**Fig. 13.70** HTMLReport

### Include Text

You can enter text directly into the report editor. Line breaks and tabs are applied. To format, in addition to bold and italic, you can also use cascading style sheets (e.g. in connection with the HTML `<SPAN>` tag). HTML formatting tags are also possible. Enter the text like in Fig. 13.71 below `## Overview`.

```
## Overview
* Project: **Chemical Threatment**
* Finish-Date: <i>01.01.2015</i>
* Project-Manager: <span style="color: red; font-weight:bold">John Doe</span>
```

Fig. 13.71 Text in HTMLReports

This is displayed in the report as in Fig. 13.72.

The screenshot shows a section of an HTML report titled "General Information". Below it is a section titled "Overview". Inside the "Overview" section, there is a bulleted list of project details:

- Project: **Chemical Threatment**
- Finish-Date: *01.01.2015*
- Project-Manager: **John Doe**

Fig. 13.72 HTMLReport

### Model screenshot

You insert a screenshot of the model with:

```
><[current,*]
```

Thereby, current is the current frame.

### Show standard statistics of the objects

To represent the standard statistics of all SingleProc objects, you would add the following:

```
[.MaterialFlow.SingleProc*]
```

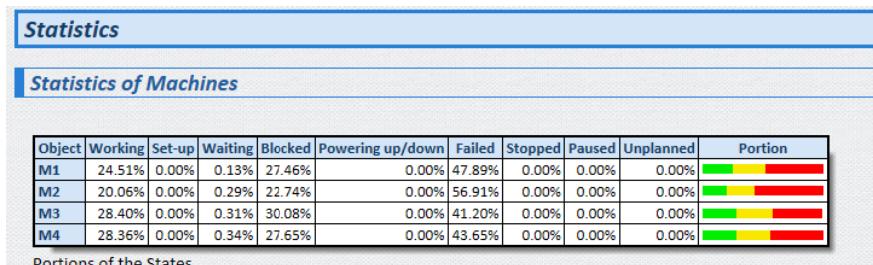
Example: Complete the report using the following elements:

```
#Statistics
```

```
##Statistics of Machines
```

```
[.MaterialFlow.SingleProc*]
```

The report is now showing a table with the default statistics (Fig. 13.73).



**Fig. 13.73 HTMLReport**

Calculated expressions are enclosed in [= ]. They can be inserted in the text. Example: You want to show the statistical values of M1. For this purpose the following text would be necessary:

```
##Statistics of M1
* Portion working: [=round(M1.statWorkingPortion*100,2)] %
* Portion waiting: [=round(M1.statWaitingPortion*100,2)] %
-- and so on
```

Charts and tables are simply inserted by an object reference into the report such as: [chart].

Method texts can be inserted with the help of the expression:

```
<pre> [= ref (method).program] </pre>.
```

### Images in the HTMLReport

You can select each icon, which is stored in an object for display in the report. Click on the second icon from the right in the report editor toolbar. In the following dialogs, you can select the object and the icon.

# Chapter 14

## Data Exchange and Interfaces

Plant Simulation provides several interfaces for import and export data.

### 14.1 DDE with Plant Simulation

Dynamic data exchange (DDE) allows accessing another program. A Windows program that makes DDE functionality available connects to the system under a server name and provides various topics. You can establish a connection to this program by providing the specified server name and a valid theme. You may use the established channel for certain transactions related to data (items) that are supported by the server. All DDE transactions are realized by SimTalk calls in Plant Simulation. You can call any method in Plant Simulation using DDE, including the methods that you have defined. Important: All participating programs must be started.

#### 14.1.1 *Read Plant Simulation Data in Microsoft Excel*

The following examples are programmed in VBA (Visual Basic for Applications). In principle, this works with Excel, Project, Word and Access. To work with DDE, you need the following commands:

**Table 14.1** VBA DDE commands

VBA/SimTalk Method	Description
DDEInitiate(<string1>, <string2>)	Opens a channel; you have to pass the name of the application (<string1>, (in Plant Simulation 12 this still is eM-Plant) and a topic (<string2>) Plant Simulation supports the following topics: System (you can request information about Plant Simulation and call SimTalk methods) Data (values of global variables can be read and written) Info (version of Plant Simulation, name of the current frame, states of the frame you can read)

**Table 14.1 (continued)**


---

DDERequest(<integer>, <string>)	Requests information from an application; you need to pass the channel number (<integer>) and an item (<string>); the passed element (item) will be queried (Note: It must be addressed absolutely.)
DDETerninate(<integer>)	Closes the channel (<integer>)
DDEPoke(<integer>, <string>, <string>, <integer>)	Writes data (variant) into the connected application (channel <integer>, element <string>, timeout<integer>)
DDEExecute(<integer>, <string>)	Sends a command (<string>) to the connected application (<integer> channel)

---

**Example: Data Exchange DDE Excel**

The following examples are programmed in VBA. In principle, this works with Excel, Project, Word, and Access. Insert a variable (name: variable; data type: integer; value: 247985) into a Plant Simulation frame (.Models.Frame). This value is to be read in Excel. Start Excel. Open the VBA Editor. Insert a new module (Insert—Module) and a new procedure (sampleDDE) in the module:

```
Public Sub sampleDDE()
Dim channel As Long
Dim value As Variant
'establish channel
channel = DDEInitiate("eM-Plant", "Data")
'read the value
value = DDERequest(channel, _
".Models.Frame.Variable")
'write the value into the table
Tabelle1.Range("A2").value = value(1)
'close channel
DDETerninate (channel)
End Sub
```

**14.1.2 Excel DDE Data Import in Plant Simulation**

Plant Simulation can also read data from DDE-enabled programs. Regarding Excel, a few requirements must be met: The Excel file must be opened. You have to specify the worksheet in Excel when opening the connection. All data will be transferred as a string. Reading data from Excel with DDE is done in three steps:

1. Establish a connection with Excel
2. Read the value
3. Close the connection

To read a value from the cell B2 in Sheet1, the following program is necessary:

```
is
channel:integer;
val:string;
```

```

do
  channel:=DDEConnect("Excel", "Sheet1");
  val:=DDERequest(channel, "Z2S2");
  --remove line break and carriage return
  val:=omit(val,strlen(val)-1,2);
  print val;
  DDEDisconnect(channel);
end;

```

**Example: Data exchange, importing a working plan from Excel via DDE**

You are to import a work plan from Excel into Plant Simulation. The work plan should have, for example, the form in Fig. 14.1 (insert a table named working\_plan into a frame in Plant Simulation).

	string 1	string 2	string 3
string	Machine	Processing time	Successor
1			

**Fig. 14.1** Example Plant Simulation table

Place an analogous table in Excel (sheet Sheet1) and enter data according to Fig. 14.2.

	A	B	C
1	Machine	Processing time	Successor
2	M1	10	M2
3	M2	20	M4
4	M4	30	M6
5	M6	10	Drain

**Fig. 14.2** Example Excel table

You are to transfer the data from Excel to Plant Simulation. The following example loads all data in the Excel sheet Sheet1 into a Plant Simulation table (working\_plan). Method: load\_workingplan

```

is
  value, address:string;
  channel, row, column:integer;
  colNext, rowNext:boolean;
do
  DDEDisconnectAll;
  --establish connection
  channel:=DDEConnect("Excel", "Sheet1");
  --column by column, until no more values available
  row:=2;
  column:=1;

```

```

rowNext:=true;
while(rowNext) loop
--start from address 1,1 column by column
column:=1;
colNext:=true;
--no value, stop both loops
adress:="R"+to_str(row)+"C"+to_str(column);
value:=ddeRequest(channel,adress);
--returns additional a line break
if value = chr(13)+chr(10) then
  rowNext:=false;
  colNext:=false;
else
  while (colNext) loop
    adress:="R"+to_str(row)+"C"+to_str(column);
    value:=ddeRequest(channel,adress);
    if value = chr(13)+chr(10) then
      colNext:=false;
    else
      working_plan[column,row-1]:=omit(value,strlen(value)-1,2);
      column:=column+1;
    end;
  end;
  row:=row+1;
end;
end;
ddedisconnect(channel);
end;

```

Note: The cell address depends on your installed language in Excel. In English, the cell address is, e.g. R1C1 in German Z1S1.

Check to see what Excel returns, if you read the value of an empty cell. In our example, Excel returns a line break and a carriage return (ASCII 10 and 13) at the end of each value. You have to remove this before writing the value into the Plant Simulation table. Within the method `load_workingPlan`, I used the method `omit (<string>, <integer>, <integer>)`.

### 14.1.3 Plant Simulation DDE Data Export to Excel

With DDEPoke, you can write data into other applications. The method has four transfer parameters: channel, topic, value, and a maximum timeout. To write data from a variable (variable) to Excel in the sheet Sheet2, cell B2, the following program would be necessary:

```

is
  channel:integer;
do
  channel:=DDEConnect( "Excel" , "Sheet2" );
  --write in sheet2 B2
  DDEPoke(channel, "Z2S2",to_str(variable),1000);
  DDEDDisconnect(channel);
end;

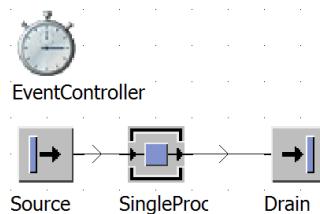
```

#### 14.1.4 Plant Simulation Remote Control

You can call Plant Simulation commands with the function DDEExecute (channel, command) in Excel.

##### Example: DDE Remote Control

For the following example, you must enable macros in Excel. Create a simple frame like in Fig. 14.3.



**Fig. 14.3** Example frame

Suppose you want to control the frame .Models.Frame from an Excel file. The simulation is to start and stop with buttons in Excel. Add two buttons to an Excel spreadsheet (Fig. 14.4).

	A	B	C
1			
2		Start simulation	
3			
4			
5			
6		Stop simulation	
7			

**Fig. 14.4** Excel buttons

The event handling for starting the simulation might appear as follows.

```

Private Sub CommandButton1_Click()
Dim channel As Long
'Establish connection

```

```

channel = DDEInitiate("eM-Plant", "System")
'Execute command
DDEExecute channel, _
  ".models.frame.eventController.start"
'close connection
DDETerminate (kanal)
End Sub

```

Create the event handling for stopping the simulation in Excel.

### 14.1.5 DDE Hotlinks

A simple way of exchanging data is using links. The values of the links are automatically updated. Within DDE, these links are called hotlinks. To make it work, you must first enable the corresponding global variable for the hotlink.

#### Example: DDE hotlinks

Insert a global variable named variable into a frame. The variable is to indicate the number of completed parts. You need a method (e.g. entrance control drain), which might appear as follows:

```

is
do
  variable:=drain.statNumIn;
end;

```

You have to enable the variable for DDE Hotlinks. Select the option Support DDE Hotlinks in the dialog of the variable on the tab Communication (Fig. 14.5).

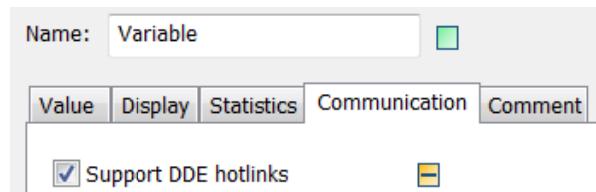


Fig. 14.5 DDE hotlink

In Excel, you can now set a hotlink to the variable in a table cell (Fig. 14.6).

$f_x$	<code>='eM-Plant'!Data!.models.frame.variable'</code>		
C	D	E	F
output		12399	

Fig. 14.6 DDE hotlink Excel

Once the value of the variable changes, it will update the value of the cell automatically.

## 14.2 The COM Interface

Siemens offers the possibility to access and remotely control Plant Simulation from other programs via an interface. It is possible via the COM interface to access values in Plant Simulation, invoke methods and react to events that are fired from Plant Simulation. In the following two examples, Plant Simulation is accessed via Excel. The programming is performed in VBA.

### 14.2.1 Read Data from Plant Simulation

In the first step, you need to read the contents of a Plant Simulation table into an Excel table. To do this, enter a button in an Excel spreadsheet and specify an event handler for it. In the VBA editor, you must first set a reference to the Plant Simulation Library (VBA editor—Tools—References). Place a check mark in the Tecnomatix Plant Simulation Type Library (Fig. 14.7).

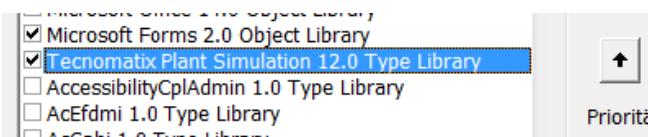


Fig. 14.7 Reference VBA editor

Note: If you have installed a 32-bit version of Microsoft Office, then you must also install the 32-bit version of Plant Simulation. The 32-bit Office cannot communicate with the 64-bit version of Plant Simulation and vice versa.

The contents of the library are visible on the Object Browser (key F2). The name of the Plant Simulation library is eMPlantLib. It contains one class: RemoteControl (see Fig. 14.8).

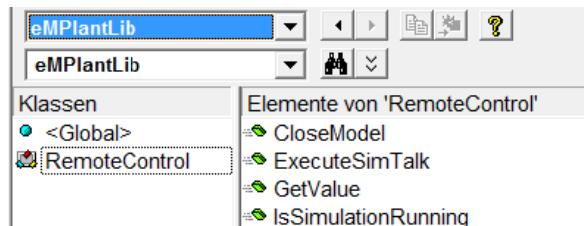


Fig. 14.8 VBA object catalogue

For the first step, you need the methods from Table 14.2:

**Table 14.2** COM methods

Method	Description
<rc>.LoadModel(<path>)	Opens a Plant Simulation file
<rc>.getValue(<ps_path>)	Reads from the open Plant Simulation model values from the path ps_path
<rc>.quit	Closes Plant Simulation, changes are not saved

The method `getValue` allows access to attribute values, tables, lists, return values of methods, etc. The sequence of a COM access is as follows:

1. Create an instance of `RemoteControl`
2. Open the Plant Simulation file
3. Read data
4. Close Plant Simulation

Write all the values from a table (.models.frame.tabellen) in the table Sheet1 of the Excel file. The following VBA programming is necessary for this purpose:

```
Sub test()
Dim ps As RemoteControl
Dim x As Integer
Dim y As Integer
Dim i As Integer
Dim k As Integer
Set ps = New RemoteControl
'change the path
ps.LoadModel ("d:\com.spp")
'dimensions of the Plant Simulation table
x = ps.GetValue(".models.frame.tabellen.xDim")
y = ps.GetValue(".models.frame.tabellen.yDim")
'read all values
For i = 1 To y
    For k = 1 To x
        Tabellen1.Cells(i, k).Value =
            ps.GetValue(".models.frame.tabellen[" +
            CStr(k) + "," + CStr(i) + "]")
    Next
Next
'close Plant Simulation
ps.Quit
End Sub
```

### 14.2.2 Write Data, Call Methods, Respond to Events

With the COM library, you can, in addition to reading values, change values in Plant Simulation, invoke methods and respond to events from Plant Simulation. The following scenario is intended to serve as an example: In Excel data are defined for the simulation. By a click on a button, these data are entered in Plant Simulation and the simulation begins. After the end of the simulation, the simulation results are entered into the Excel spreadsheet.

#### VBA Class Module—Event Handling

The event handler must be created in a VBA class module (as an internal method). This requires a special approach. First, create a class module in the VBA editor (Insert—Class module). Rename the class as "ps" (Properties Window—(name)). In the class, you must define an event handler. All events will be sent to this object. Therefore, write the following statement in the head of the class module:

```
Public WithEvents ps As eMPlantLib.remotecontrol
```

The object variable will be initialized via the constructor. To do this, add the following method to the class module:

```
Private Sub Class_Initialize()
    Set ps = New remotecontrol
End Sub
```

Create the table from Excel (Sheet1).

	A	B	C	D	E	F	G
1			results				
2		waiting	working	blocked		process time (sec)	
3	M1	0%	0%	0%	M1		
4	M2	0%	0%	0%	M2		10
5							
							start simulation

Fig. 14.9 Example Excel table

You also need a frame in Plant Simulation like in Fig. 14.10.

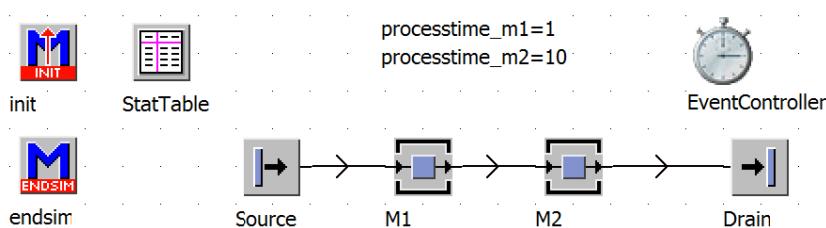


Fig. 14.10 Example frame

Machines M1 and M2 each have an availability of 95 per cent and an MTTR of one minute. Set the interval of the source to zero. Create the table StatTable according to Fig. 14.11.

	string 0	real 1	real 2	real 3
string		waiting	working	blocked
1	M1			
2	M2			

**Fig. 14.11** Table StatTable

The following procedure is to be programmed:

1. By clicking on the button Start Simulation, Plant Simulation is started and the model is loaded; the processing times of M1 and M2 are transferred to Plant Simulation. The simulation is initialized (reset) and started.
2. The init method sets the process times of the machines.
3. EndSim writes the statistical data into the table StatTable and sends an event to Excel that the simulation is complete.
4. The event handler in Excel writes the results of the simulation into the Excel spreadsheet and terminates the simulation (and closes Plant Simulation).

The following methods of the COM interface are required in VBA for the next steps (Table 14.3):

**Table 14.2** Methods of the class RemoteControl

Methods	Description
<rc>.LoadModel(path)	Opens a Plant Simulation model
<rc>.getValue(ps_path)	Reads from the open Plant Simulation Model values with the path ps_path
<rc>.setValue ps_path, value	Sets values in Plant Simulation
<rc>.ResetSimulation (eventController)	Resets the simulation
<rc>.startSimulation (eventController)	Starts the simulation
<rc>.quit	Closes Plant Simulation, changes are not saved

For 1.), assign a macro to the button in Excel: Insert the following programming (you will have a different name for the method; note the declaration of pss outside of the method).

```
Dim pss As ps
Sub Schaltfläche1_KlickenSieAuf()
Set pss = New ps
pss.ps.LoadModel ("D:\com_1_en.spp")
```

```

'set values
pss.ps.SetValue ".Models.Frame.processtime_m1",_
    Range("G2").Value
pss.ps.SetValue ".Models.Frame.processtime_m2",_
    Range("G3").Value
'start simulation
pss.ps.ResetSimulation(".Models.Frame.EventController")
pss.ps.StartSimulation
".Models.Frame.EventController")
End Sub

```

To 2.) Init method of the Plant Simulation model:

```

is
do
    m1.procTime:=num_to_time(processtime_m1);
    m2.procTime:=num_to_time(processtime_m2);
end;

```

To 3.) Method endSim of the Plant Simulation model:

```

is
do
    --write statistics
    statTable[1,1]:=m1.statWaitingPortion;
    statTable[1,2]:=m2.statWaitingPortion;
    statTable[2,1]:=m1.statworkingPortion;
    statTable[2,2]:=m2.statWorkingPortion;
    statTable[3,1]:=m1.statBlockingPortion;
    statTable[3,2]:=m2.statBlockingPortion;
    fireSimTalkMessage("finish");
end;

```

For 4.), the event handler must be programmed in the VBA class module. To do this, select the module sheet in the head with ps on the left and SimTalkMessage on the right. The method ps\_SimTalkMessage takes as arguments the string that you pass via SimTalk with **fireSimTalkMessage**.

```

Private Sub ps_SimTalkMessage(ByVal text As String)
Dim x As Integer
Dim y As Integer
Dim i As Integer
Dim k As Integer
If text = "finish" Then
    x = ps.GetValue(".models.frame.statTable.xDim")
    y = ps.GetValue(".models.frame.statTable.yDim")

```

```

For i = 1 To y
    For k = 1 To x
        Tabelle1.Cells(i + 2, k + 1).Value = _
            ps.GetValue(".models.frame.statTable[" +
            CStr(k) + "," + CStr(i) + "]")
    Next
Next
ps.Quit
End If
End Sub

```

## 14.3 The ActiveX Interface

ActiveX allows access to the COM interfaces of different applications. Thus, functions of other programs can be integrated in Plant Simulation. A large number of programs provide COM libraries, which can be integrated via the ActiveX interface of Plant Simulation.

### 14.3.1 ActiveX and Excel

The first step is to integrate Excel in Plant Simulation. Using the ActiveX interface you can not only read data from Excel and write to Excel, but it also allows access to the full functionality of Excel. This is especially interesting in the area of data analysis and evaluation.

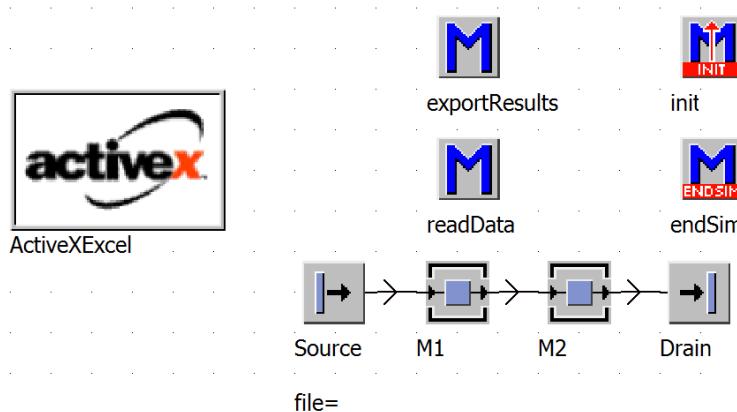
#### Example: ActiveX and Excel

Create an Excel sheet (Table1) like in Fig. 14.12.

	A	B	C	D	E	F	G	H
1	Input		Results					
2		ProcTime				working	blocked	waiting
3	M1	21			M1	1	0	0
4	M2	20			M2	0,952153	0	0,047847

Fig. 14.12 Excel Table1

In a simulation, the basic data should be read from Excel (processing time) and the results should be written back to the Excel file (statistical data). Create a frame like in Fig. 14.13.



**Fig. 14.13** Example frame

Let all the blocks retain the basic settings. The following steps are to be set up:

1. The user selects the file containing the data and the results
2. The data are read into the simulation (method `readData`)
3. At the end of the simulation, the statistical data are entered into the Excel spreadsheet

For 1.), the file should be selected via a file-open dialog. For this purpose, Plant Simulation provides the `selectFileForOpen` method. It returns the full path of the selected file. If the user selects Cancel, the method returns an empty string. The `init` method might look like this (the file open dialog will be displayed only if a path is not already set):

```

is
do
  if file="" then
    messagebox("Choose the exel-file, "+
              " which contains the data!",1,2);
    file:=selectFileForOpen("Excel-Files (*.xlsx)| "+
                           "*.*xlsx| ");
  end;
  if file /= "" then
    readData;
  else
    eventcontroller.stop;
  end;
end;

```

Excel provides all of the functionality via a top object (Excel.Application). Ensure settings in the ActiveX interface like in Fig. 14.14 .

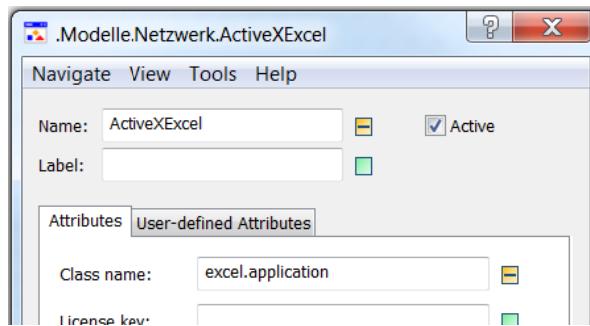


Fig. 14.14 ActiveX interface

With View—Type info, you get an overview of the attributes and methods of the class. You get a better overview, however, in the Object Browser in the VBA editor in Excel and in VBA help.

2.) To read the data, you must open Excel, load the file and transfer the data from the Excel table to Plant Simulation (method readData).

```
is
do
  --open file
  ActiveXExcel.workbooks.open(file);
  --transfer data
  M1.procTime:=
  ActiveXExcel.ActiveWorkbook.
    sheets("Table1").Range("B3").value;
  M2.procTime:=
  ActiveXExcel.ActiveWorkbook.
    sheets("Table1").Range("B4").value;
  --close file
  ActiveXExcel.ActiveWorkbook.close;
end;
```

Access to all cells of an Excel file is ensured by a central object: Active Workbook. The Workbook object provides a collection of worksheets (Sheets). The easiest access to a spreadsheet is by using its name (e.g. Sheets ("Sheet1")). Individual cells or ranges can be accessed via the Range object, which is provided by the sheet object. Thus, for example, the access to the value of cell B2 of Sheet1 takes place with:

ActiveXExcel.ActiveWorkbook.Sheets("Sheet1").Range("B2").value.

3.) At the end of the simulation, the statistical data are entered into the Excel spreadsheet and the changes in the Excel file are saved (method exportResults called by endSim):

```
is
do
  --write results directly to Excel
  --open file
```

```

ActiveXExcel.workbooks.open(file);
--write statistical data
ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("F3")
).value:=
M1.statWorkingPortion;
ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("G3")
).value:=
M1.statBlockingPortion;
ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("H3")
).value:=
M1.statWaitingPortion;

ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("F4")
).value:=
M2.statWorkingPortion;
ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("G4")
).value:=
M2.statBlockingPortion;
ActiveXExcel.ActiveWorkbook.sheets("Table1").Range("H4")
).value:=
M2.statWaitingPortion;
--save file
ActiveXExcel.ActiveWorkbook.save;
ActiveXExcel.ActiveWorkbook.close;
end;

```

If you need to read data from a large number of machines using Excel, the following procedure may be of interest:

- locate the machine in the Excel file
- read the corresponding data from the Excel table

From Plant Simulation, you can search in an Excel spreadsheet using the VBA command Range.Find (the result is a cell reference). With range.offset (row, column), you can address cells relative to a cell and read their content.

Insert a table (machines) in Plant Simulation with the content according to Fig. 14.15.

	string 1	string 2
1	M1	
2	M2	

**Fig. 14.15** Table machines

An alternative method for reading the machine data from Excel might look like this:

```

is
i:integer;

```

```

xls:any;
cell:any;
do
  --open file
  ActiveXExcel.workbooks.open(file);
  xls:=ActiveXExcel.ActiveWorkbook.Sheets("Table1");
  for i:=1 to machines.yDim loop
    --look for the machine-name in excel
    if xls.Range("A1:C1000").find(machines[1,i]) /=
      void then
      cell:=xls.Range("A1:C1000").find(machines[1,i]);
      str_to_obj(machines[1,i]).procTime:=
        cell.offset(0,1).value;
    else
      messagebox("Data source does not contain "+ 
        machines[1,i]+"!",1,13);
    end;
  next;
  ActiveXExcel.ActiveWorkbook.close;
end;

```

You could also program an alternative Export method with the help of the machines table and the Excel OFFSET method:

```

is
  startCell:any;
  i:integer;
do
  ActiveXExcel.workbooks.open(file);
  startCell:=

  ActiveXExcel.ActiveWorkbook.sheets("Tabelle1").Range("F
3");
  --start writing from here
  for i:=1 to machines.yDim loop
    startCell.offset(i-1,0).value:=
      str_to_obj(machines[1,i]).statWorkingPortion;
    startCell.offset(i-1,1).value:=

    str_to_obj(machines[1,i]).statBlockingPortion;
    startCell.offset(i-1,2).value:=
      str_to_obj(machines[1,i]).statWaitingPortion;
  next;
  ActiveXExcel.ActiveWorkbook.save;
  ActiveXExcel.ActiveWorkbook.close;
end;

```

### 14.3.2 ActiveX Data Objects (ADO, ADOX)

The ActiveX Data Objects (ADO) allows access to a large number of data sources through a standard interface (e.g. ACCESS or SQL Server). ADO also allows database operations on computers where no database system is installed. In the following, the most important works in a database will be shown:

- Create databases
- Create database tables
- Insert data
- Search data
- Change and delete data

#### 14.3.2.1 Creating a New Access Database

The object model ADOX contains objects, properties and methods to create, edit and view the structure of databases, tables and queries. ADOX was developed so that it works with all OLEDB providers. To create a new database, you need to use the Create method of the catalog object. You must set up the ActiveX object for use of the ADOX.catalog as in Fig. 14.16.

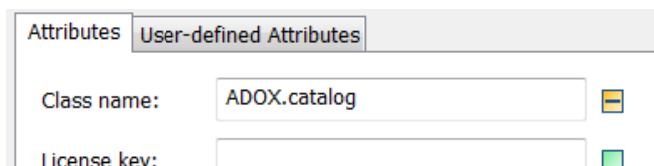


Fig. 14.16 ActiveX object

The following example creates a new database. You must specify the provider and the path of the database. The path of the database is determined via a file-save-as menu.

```
is
do
--pick up file-name
dbFile:=selectFileForSave("Database files "+
    "(*.mdb, *.accdb) | "+
    "* .mdb; *.accdb| |");
if dbFile /= "" then
    --the database should not exist
    catalog.Create("Provider="+
        "Microsoft.ACE.OLEDB.12.0;Data Source="+
        dbFile);
end;
end;
```

### 14.3.2.2 Integrating the Necessary Libraries

The ActiveX object gives you access to the ADO objects. You can find the ActiveX objects in the class library under standard objects. For each top object, you need one ActiveX object. Access to a database with ADO is done using a connection string. The connection string is database-dependent. For accessing an ACCESS database, the connection string is:

"Microsoft.ACE.OLEDB.12.0;Data Source=xxxx" (starting from Microsoft Office 2007). If the driver on the target system does not exist (OLEDB—driver), you can download it from Microsoft:

<http://www.microsoft.com/de-de/download/details.aspx?id=13255> (even for 64-bit systems).

Note: You cannot access the 32-bit driver from a 64-bit application and vice versa. This means that if you use a 32-bit Microsoft installation, then you can only access the Access database from the 32-bit version of Plant Simulation.

The process of working with ADO is usually the following:

1. Establish a connection
2. Open the database (connection string)
3. Perform actions
4. Close database

Insert an ActiveX object and a method in a frame. Name the ActiveX object as "Connection" and then open the ActiveX object by double-clicking it. In the field Class Name, enter the name of the class that you want to include. The next class you need is ADODB.Connection. With View—Type info, you get an overview of the available properties and methods.

### 14.3.2.3 Creating a Database Table

You create a table using ADO in three steps:

1. Open a connection to the database in which you want to add the table
2. Create using the Create statement a new table
3. Close the connection

Example: You want to create a database of articles (db.mdb). In the database articel.mdb, there should be a table named data with the following fields:

Name	Data type
ID	INTEGER
nr	CHARACTER
name	CHARACTER
description	TEXT
price	REAL

```

is
  sql:string;
do
  connection.provider:-
  "Microsoft.ACE.OLEDB.12.0";
  connection.open(dbFile);
  sql:="CREATE TABLE data (ID AUTOINCREMENT" +
    " PRIMARY KEY,nr CHARACTER, name CHARACTER, " +
    " description TEXT,price REAL)";
  connection.Execute(sql);
  connection.close;
end;

```

The field definition consists of field name and data type. The following data types are defined for ADODB (e.g. ACCESS 2007 and thereafter): BINARY, BIT, TINYINT (one byte), MONEY, DATE, TIME, UNIQUEIDENTIFIER, REAL (four bytes), FLOAT (eight bytes), SMALLINT (two bytes), INTEGER (four bytes), DECIMAL (17 bytes), TEXT (two bytes per character, maximum 2.14 GB) IMAGE (binary for OLE objects) CHARACTER (two bytes per character, maximum 255 characters).

If a table with the same name already exists, a runtime error is generated.

#### 14.3.2.4 Insert Data into a Table, Add Records

If a table exists, you can enter records. This works in the following sequence:

1. Set up a connection to the database
2. Insert data using the INSERT statement
3. Close the connection

Example: There are a number of Plant Simulation data to be transferred to Access. Create a table in Plant Simulation according to Fig. 14.17.

	string 1	string 2	string 3	real 4
string	nr	name	description	price
1	A123	W1	shaft	34.67
2	A124	W2	shaft	67.98
3	A125	R1	gear	123.98

**Fig. 14.17** Table article

The table rows are successively packed with INSERT statements and sent to the database.

```

is
  sql:string;
  i:integer;

```

```

do
  --establish connection
  connection.provider:-
  "Microsoft.ACE.OLEDB.12.0";
  connection.open(dbFile);
  --create sql statement
  for i:=1 to article.yDim loop
    sql:="INSERT INTO data" +
      " (nr,name,description,price) " +
      "values('" +
      article[1,i]+",'" +
      article[2,i]+",'" +
      article[3,i]+",'" +
      to_str(article[4,i]))";
    connection.execute(sql);
  next;
  connection.close;
end;

```

#### 14.3.2.5 Find and Display Records

For searching and displaying data, you need the ActiveX object ADODB.Recordset. The Recordset provides search results row (record) by row. Example: You want to show the article number and the price of all records. Create a table in Plant Simulation like in Fig. 14.18.

	integer	string	real	string
string	ID	nr	price	
	1			4

Fig. 14.18 Price list

Therefore, the following SQL statement is required:

```
"SELECT id,nr,price FROM data"
```

This SQL statement is passed at the creation of the Recordset object as the first parameter. As a second, you must pass the connection string of the data provider.

```

is
  sql:string;
  i:integer;
  connectionID:any;
do
  priceList.delete;
  sql:="SELECT id,nr,price FROM data";
  --establish connection

```

```

connection.provider:-
  "Microsoft.ACE.OLEDB.12.0";
connection.open(dbFile);
--create new recordset
recordset.open(sql,connection.connectionString);
while not recordset.eoF loop
  --write all to pricelist
  priceList.writeRow(1,priceList.yDim+1,
    recordset.fields.item(0).value,
    recordset.fields.item(1).value,
    recordset.fields.item(2).value);
  recordset.moveNext;
end;
recordset.close;
connection.close;
end;

```

Through the Recordset, you can move with MoveNext. There are two positions outside of the Recordset: BOF (begin of file) and EOF (end of file). You need to check before accessing the data on the Fields collection, if the cursor is located at the position EOF. The data are provided via a Fields collection (Fields.Item (i) .Value). After reading the data, the Recordset is closed with Close.

#### 14.3.2.6 Updating Data

The updating of data can be done through the execute method of the Connection object using SQL (e.g. UPDATE). On the other hand, the Recordset object supports methods for the changing of data.

Example: You want to update the database with the actual data from the price. For this purpose, the following programming is required (version Connection.Execute):

```

is
  sql:string;
  i:integer;
do
  connection.provider:-
  "Microsoft.ACE.OLEDB.12.0";
  connection.open(dbFile);
  --store data from priceList to the database
  for i:=1 to priceList.yDim loop
    sql:="Update data set price=" +
      to_str(priceList[3,i])+" WHERE ID=" +
      to_str(priceList[1,i]);
    connection.Execute(sql);
  next;
  connection.close;
end;

```

### Version updatable Recordset

If you use a scrollable and updatable Recordset object in a query, you can make changes in the data source directly through the Recordset object. All you need is to assign a new value via the Value property to the Field objects in the Fields collection of the Recordset and to write these changes with the update method of the Recordset object in the database. What follows might look like a method that updates a field in a record, which was determined by an SQL select statement.

```

is
  sql:string;
  i:integer;
do
  connection.provider:-
    "Microsoft.ACE.OLEDB.12.0";
  connection.open(dbFile);
  for i:=1 to priceList.yDim loop
    --get records
    sql:="SELECT price from data WHERE ID="
      + to_str(priceList[1,i]);
    recordset.open(sql,connection.connectionString,
      1,3);
    --update
    recordset.fields(0).value:=priceList[3,i];
    --save
    recordset.update;
    recordset.close;
  next;
  connection.close;
end;

```

## 14.4 The File Interface

Using the file interface, you can access text files to read, delete, and overwrite their contents. SimTalk provides a set of methods and attributes to access files (Table 14.4).

**Table 14.3** Attributes and methods of the File Interface

Method/Attribute	Description
<path>.fileName	Sets/writes the filename of the FileInterface
<path>.remove	Deletes the specified file
<path>.goToLine(<integer>)	Sets the pointer of the file on the specified line; thus, you can write and read line by line
<path>.eof	Returns true, if the end of file is reached
<path>.writeLn (<string>)	Overwrites the line at the current line position with the given content
<path>.readLn	Returns the content of the actual line as string

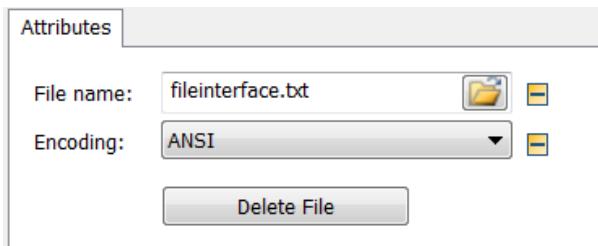
### Example: File Interface

For the example, you need an object of type Comment, one FileInterface, and two Method objects (Fig. 14.19).



**Fig. 14.19** Example frame

Type some text into the comment tab of the Comment object. This text is to be written into a file. In the Comment object, disable the option "Save the content in rich-text format." Create a file fileinterface.txt in the same folder as the Plant Simulation file. You first have to set up the file interface. Open the object FileInterface, and select or enter the settings from Fig. 14.20.



**Fig. 14.20** FileInterface

The content of the Comment object will be written to the file.

Method: writeFile:

```
is
do
  fileInterface.open;
  fileInterface.writeln(comment.cont);
  fileInterface.close;
end;
```

The contents of the file must now be displayed in the console.

Method: readFile

```

is
  i:integer;
do
  --writes content of the textfile in the console
  fileInterface.open;
  i:=1;
  fileInterface.gotoLine(i);
  while not fileInterface.eof loop
    print fileInterface.readln;
    i:=i+1;
    fileInterface.gotoLine(i);
  end;
  fileInterface.close;
end;

```

## 14.5 The ODBC Interface

The ODBC interface allows one to access ODBC data sources. You can, for example, use the ODBC interface for reading data from a database, for modifying data in databases, and for entering data from the Plant Simulation into a database. You must first add the object ODBC to the Class Library (for this, you need the Plant Simulation interface package). Select Home—Model—Manage Class library (v. 12..., Fig. 14.21).



Fig. 14.21 Manage Class Library in version 12

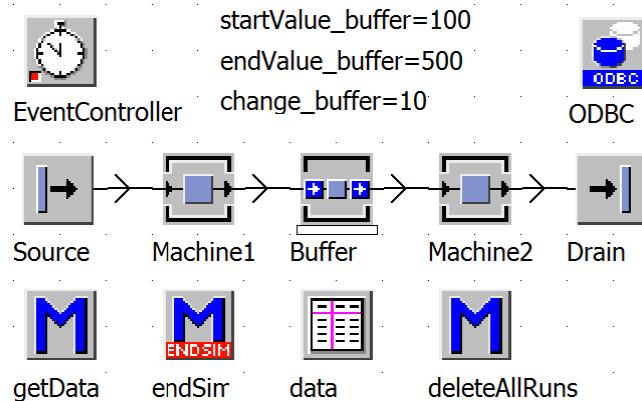
Look for the ODBC interface in the branch Information flow. Select ODBC and confirm your selection with OK.

### 14.5.1 Setup an ODBC Data Source

Before you can use the ODBC Interface, you must create a database and set the database up as an ODBC data source.

#### Example: ODBC

You are to simulate two machines and one buffer. Create the frame from Fig. 14.22.



**Fig. 14.22** Example frame

The Source produces one part (blocking) every minute. You are to save the settings and the results of the simulation into a database. Insert a database into Access and save the database under production\_database (.mdb). The database will initially contain two tables:

Table Elements:

Field name	Data type	Commit
ID	Auto value	Primary key
Name	Text	Maximum 50 characters
Processing_time	Number	Long integer (seconds)
Availability	Number	Double
MTTR	Number	Long integer (seconds)
Capacity	Number	Long integer (default 1)

Table simulation\_runs:

Throughput, average exit interval (cycle time), and output as a function of buffer size are to be recorded.

Field name	Data type	Commit
ID	Auto value	Primary key
Buffer_capacity	Number	Long integer
Throughput	Text	
Exit_interval	Text	
Output	Number	Long Integer

Type data in the table elements according to Fig. 14.23.

ID	Name	Processing_time	Availability	MTTR	Capacity
1	Machine1	50	75	18000	1
2	Buffer	0	100	0	100
3	Machine2	45	50	36000	1

**Fig. 14.23** Access data

Set up the database as an ODBC data source. Under Microsoft Windows, you select Control Panel—Administrative Tools—Data sources (ODBC). Click the tab System DSN and then click Add. Select the Access database driver and click Finish. In the next window, type in a name for the Data source (production\_database) and select the database. The database is now accessible on the data source name (DSN).

### 14.5.2 *Read Data from a Database*

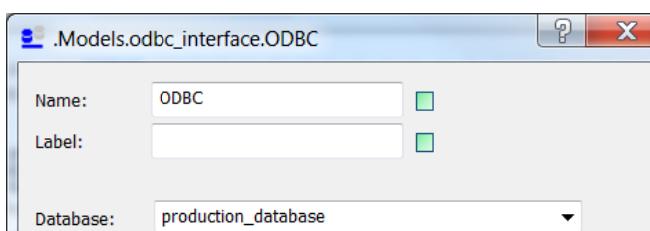
You need the commands from Table 14.5 for working with the ODBC interface.

**Table 14.4** ODBC interface methods

Method	Description
<path>.login (<string>, <string>, <string>)	Establishes a connection to the database; you need to pass the database name, the login name and password
<path>.logout	Closes the database connection and releases the memory
<path>.sql(<table>, <string>)	Sends an SQL statement to the database; the results of the query are written into the passed table

#### Example: ODBC (continuation)

Open the ODBC object. Enter the name of the ODBC data source into the field database (Fig. 14.24).



**Fig. 14.24** ODBC setting

Confirm your changes. No other settings are required in this window.

Example: You are to read all data from the database table elements into the table data. Method getData:

```

is
  sql:string;
do
  --login
  ODBC.login("production_database", "", "");
  --form sql-command
  sql:="SELECT * FROM Elements";
  -- send sql command,
  --Plant Simulation writes the result
  --into the table data
  ODBC.sql(data,sql);
  -- logout
  ODBC.logout;
end;

```

Execute the method. Plant Simulation saves the query results in the table data. You can read the data from there (Fig. 14.25).

	integer 1	string 2	integer 3	real 4	integer 5	integer 6
string	id	Name	Processing_time	Availability	MTTR	Capacity
1	1	Machine1	50	75.000000	18000	1
2	2	Buffer	0	100.000000	0	100
3	3	Machine2	45	50.000000	36000	1

Fig. 14.25 Table data

The setting of the data using the table values may now look like this.

Method getData:

```

is
  sql:string;
  i:integer;
  element:object;
do
  --login
  ODBC.login("production_database", "", "");
  --create sql-statement
  sql:="SELECT * FROM Elemente";
  -- send sql
  ODBC.sql(data,sql);
  -- logout
  ODBC.logout;
  for i:=1 to data.Ydim loop
    element:=str_to_obj(data[2,i]);
    element.procTime:=data[3,i];
    element.failures.failure.availability:=data[4,i];
    element.failures.failure.mttr:=data[5,i];

```

```

element.capacity:=data[6,i];
next;
end;

```

Note: Before executing the method, insert a failure to all objects (without any settings). The default name of the failure is "failure." Then, the method works without error messages.

### 14.5.3 Write Data into a Database

Writing data in a database is analogous to reading data. You will establish a connection to the database, send an SQL statement, and close the connection after completing work with the database. For entering new records, use the SQL command INSERT; for modifying existing records, use UPDATE.

#### Example: ODBC (continuation)

Suppose you want to automatically execute a series of simulation runs. The aim is to increase the size of the buffer by the value startvalue\_buffer up to a value endvalue\_buffer to a size change\_buffer. Each variant will be simulated (setting in the EventController). After completing a run (endSim), the values are to be entered into a new row in the table simulation\_runs.

Insert the global variables from Fig. 14.26 into the frame, and enter the initial values.

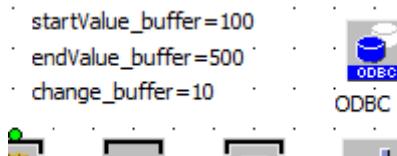


Fig. 14.26 Global variables

For entering the data into the table, use the SQL command INSERT. The syntax of this command is:

INSERT INTO table (field1, field2, field3 ...) VALUES (value1, value2, value3 ...).

Strings must be specified in the SQL statement in single quotation marks.

The method init might appear as follows:

```

is
  sql:string;
do
  --insert data
  sql:="INSERT into simulation_runs "+ 
    "(Buffer_capacity, " +
    "Throughput, Exit_interval, Output) Values (" +
    to_str(buffer.capacity)+", " +

```

```

      """+to_str(drain.statavgLifeSpan)+"', "+
      """+to_str(drain.statavgExitInterval)+"', "+
      to_str(drain.statnumOut)+"");
ODBC.login("production_database", "", "");
ODBC.sql(sql);
ODBC.logout;
--if buffer capacity not endvalue_buffer increase
-- and restart the simulation
if buffer.capacity < endvalue_buffer then
    buffer.capacity:=buffer.capacity+change_buffer;
    eventController.reset;
    eventController.start;
end;
end;

```

#### 14.5.4 Delete Data in a Database Table

You can delete values from database tables with the SQL command DELETE. If you do not restrict deletion by a WHERE clause, all data in the table will be deleted.

##### Example: ODBC (continuation)

You want to delete all data in the table simulation\_runs with a SimTalk method. The SQL command is

DELETE FROM simulation\_runs.

The method deleteAllRuns might appear as follows:

```

is
    sql:string;
do
    sql:="DELETE FROM simulation_runs";
    ODBC.login("production_database", "", "");
    ODBC.sql(sql);
    ODBC.logout;
    --buffer capacity reset
    buffer.capacity:=startvalue_buffer;
end;

```

#### 14.5.5 SQL Commands

The database interprets all text information except the commands in SQL, which you define as a table or column name. You must set all text values within a SQL statement within single quotation marks. The databases support different ranges of SQL statements; the selection below is the lowest common denominator. Access only supports a small fraction of the currently valid SQL syntax. Square brackets mean in the following syntax description that the element is optional. Curly

brackets denote selection choices (separated by "|"). You can usually use only one. Alternative values are separated by "/". The order of the elements is mandatory.

## SELECT

```
SELECT [DISTINCT | ALL]
Select_columns, ...
[FROM table
[WHERE where_definition]
[ORDER BY { column number | col. name | formula }
[ASC / DESC]
```

A general query has the following form:

```
SELECT column FROM tableName WHERE column="value"
```

To show all columns:

```
SELECT * FROM tableName WHERE column="value"
```

To show all columns and all records:

```
SELECT * FROM tableName
```

If you want to sort the results of a query, you can insert an ORDER BY clause. You can specify the column position (starting with 1), column names, or an arithmetic expression for calculating the column position. The default sort order is ascending (ASC); to sort in descending order, you must use DESC.

Example: You want to display all records from the table elements order by name. The SQL command must appear as follows:

```
SELECT * FROM Elements ORDER BY Name
```

Often, it is expected that a word will be found, even if it is contained within another word or string. Sample: If you search for "Machine", both machines should be found. In SQL, the LIKE expression is used for this purpose. LIKE compares two strings. If they match, the record is included in the search results. Instead of using complete strings to compare, you can also use the so-called wild cards:

- \_ one character
- % for an indefinite number of characters

Example:

```
SELECT * FROM Elements WHERE Name LIKE 'Machine%' ORDER BY Name
```

## INSERT (Insert New Records)

```
INSERT [INTO] table [(column names, ...)]
VALUES (value1, ...)
```

The simplest form is

```
INSERT INTO tableName VALUES (value1,'value2',value_n)
```

The values in parentheses must form a complete data set (number and data types); otherwise, an error occurs.

### **UPDATE (Change Data)**

```
UPDATE Table_name
```

```
    SET column_name1=value1, [column_name2=value2, ...]  
    [WHERE where_definition]
```

UPDATE updates columns in table rows with new values. The SET clause determines which columns are to be changed and which value they receive. The WHERE clause, if present, determines which rows will be changed. If it is missing, all rows will be changed.

Example: You want to change the processing time of Machine1 to 100 seconds.

SQL expression:

```
UPDATE Elements SET Processing_time=100 WHERE  
Name='Machine1'
```

### **DELETE**

```
DELETE FROM tbl_name
```

```
    [WHERE where_definition]
```

DELETE deletes rows from the table tbl\_name, which meet the condition in the where\_definition and returns the number of deleted rows. If you use the DELETE statement without a WHERE clause, all rows in the table will be deleted.

Example: You want to delete the buffer in the table Elements:

```
DELETE FROM Elements WHERE Name='Buffer'
```

## **14.6 SQLite Interface**

From version 10 onward, Plant Simulation has an integrated database engine (SQLite). For general information, visit: <http://www.sqlite.org>. But Plant Simulation does not supply database management software. If you want to view the contents of the database or even to create databases or tables, then you must do everything with the help of SQL. It is easier, however, if you consider using a separate program.

### ***14.6.1 Create Databases and Tables***

In principle, the work with SQLite runs similarly as with the ODBC interface.

You establish a connection to the database, formulate an SQL statement, send the SQL statement to the database and then evaluate the feedback from the database. One special feature of SQLite is that you can create databases in memory and on disk (as a separate file). In memory, databases have the advantage that queries (also writing access) are executed very quickly; the disadvantage is that the data will be lost if Plant Simulation crashes. Simple software to manage SQLite databases can be found at <http://sourceforge.net/projects/sqlitebrowser/>. You need such software to create the database itself (file version) and to create the tables comfortably. For the generation of tables, you need the following SQL command:

```
CREATE TABLE tbl_name [(create_definition, ...)]
create_definition: col_name type [AUTO_INCREMENT]
[PRIMARY KEY] ...
```

The following data types are used in SQLite:

**Table 14.5** SQLite data types

Data type	Descripton
INTEGER	Integer values
REAL	eight-byte floating-point number
TEXT	Unicode text (UTF-8 or UTF-16)
BLOB	Data block, the data is stored as they are read

To insert a new table in a SQLite database, you would do the following:

1. Add an SQLite interface to your frame and choose your database file (or leave the default setting ":memory:" for an in-memory database)
2. Formulate an SQL statement
3. Connect to the database (open), run the SQL statement (exec) and close the connection (close)

It lacks most of the data types supported by Plant Simulation. But you can save most of the data as text and convert it again after the reading back into the respective data types (e.g. object references, time and date).

### **Example: SQLite**

You want to create a table named *inventory*. The table should have a consecutive index (*id*) as the primary key, a column for the x- and a column for the y-coordinate in the store, a column for the name of the part, and a column for the object reference of the stored part. The table structure in SQLite must appear as follows:

Field	Data type
ID (AUTO_INCREMENT, PRIMARY KEY)	INTEGER
X	INTEGER
Y	INTEGER
Name	TEXT
Partl	TEXT

To create the table, the following SQL command is necessary:

```
CREATE TABLE inventory (ID INTEGER AUTO_INCREMENT
PRIMARY KEY, x INTEGER, y INTEGER, name TEXT,
part TEXT)
```

To test this, first create an SQLite database (e.g. SQLite Browser File—New). The default file extension of SQLite is db3. Create a frame with an SQLite interface and a method. Select the database file in the SQLite interface. The following method creates the table inventory in the database:

```
is
  sql:string;
do
  sqlite.open;
  sql:="CREATE TABLE inventory (ID"+
    " INTEGER AUTO_INCREMENT PRIMARY KEY, "+
    " x INTEGER, y INTEGER, name TEXT, part TEXT)";
  sqlite.exec(sql);
  sqlite.close;
end;
```

You can check the result in the SQLite browser.

### 14.6.2 Working with SQLite Databases

In contrast to the ODBC interface, SQL statements of the SQLite interface are compiled before execution. This brings significant speed advantages in execution. A standard SQL SELECT query with the SQLite interface looks like this:

1. You connect to the database (open)
2. The SQL statement is translated (prepare)
3. The SQL statement is executed and the pointer is set to the first row of the result (step)
4. Repeated call of the method step to get to the next line
5. The columns in the query result row is read
6. The memory is released by calling finalize
7. The database connection is closed (close)

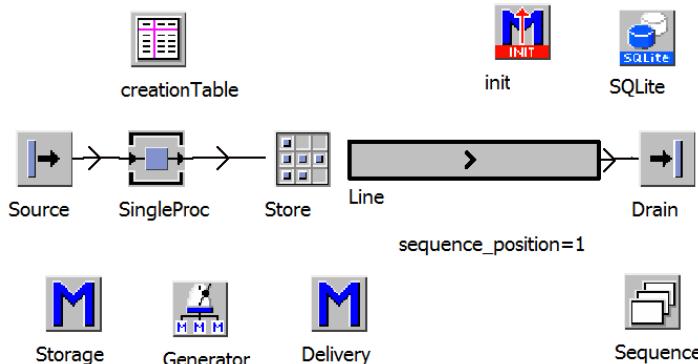
You need the following commands for working with SQLite (Table 14.7):

**Table 14.6** SQLite commands

Method	Description
<path>.open	Connects to the SQLite database; select the database (file) in the SQLite interface
<path>.prepare(<string>)	Compiles the SQL statement; SQL errors are displayed in SimTalk debugger
<path>.step	Executes the prepared SQL statement; with repeated calls of step you can move forward within the query result; step returns true if the cursor is on a row inside the result, or false if there is no result or the position of the pointer is after the last row of the result
<path>.resetStatement	Resets the SQL query; you can prepare a new query
<path>.finalize	Deletes all data in memory (compiled statement and result)
<path>.close	Closes the connection to the database
<path>.exec(<string>)	The method exec executes one after the other the commands prepare, finalize and step; use this method if you execute SQL statements from which you expect no results (such as CREATE, INSERT, UPDATE, DELETE)
<path>.getColumnCount	Number of columns in the result set
<path>.getColumnInteger(<integer>)	Returns the value of the specified column as a number; the first column has the index 0.
<path>.getColumnString(<integer>)	Returns the value of the specified column as text; the first column has the index 0

**Example: SQLite (continuation)**

You should realize an inventory database with SQLite. In a store, four different parts are stored (in batches). Deliveries are made in a sequence within the parts FIFO. The inventory management is to be mapped via an SQLite database. Create a frame like in Fig. 14.27.

**Fig. 14.27** Example frame

Insert in the class library three entities (Part1, Part2 and Part3) and customize the icons with different colors to help distinguish them. The source generates cyclically 30 Part1, 30 Part2 and 30 Part3 (Source setting: MU selection: sequence cyclically; table: creationTable, enter in the creationTable Part1, Part2 and Part3 number: 30). The singleProc has a processing time of one minute and the store has a capacity of 10,000 parts (xdim and ydim are each 100). The method storage is the exitControl of the SingleProc. The method delivery is called every minute by the generator (the first time after two hours). The line moves at a speed of 0.2 m/sec. The sequence contains as entries Part1, Part2 and Part3 (as a string).

### Part 1: Storage

The storage of the part should take place as follows: First, a free place is searched in the store. The part is moved to this place. An SQL statement is formulated that consists of an object reference of the part, the name of the part and the storage place. The SQL statement is sent to the database. All parts of the SQL statement need to be converted to strings. Storage method:

```

is
  i,k,x,y:integer;
  sql:string;
do
  --look for an empty place in the store
  x:=0;
  y:=0;
  for i:=1 to store.xDim loop
    for k:=1 to store.YDim loop
      if store.pe(i,k).empty then
        x:=i;
        y:=k;
        exitLoop 2;
      end;
    next;
  next;
  -- pack storage information into a SQL statement
  sql:="INSERT into inventory (x,y,name,part) "+
    "values ("+
    to_str(x)+","+to_str(y)+",'"+@.name+"','"++
    to_str(@)+"')";
  -- move part to the place
  @.move(store.pe(x,y));
  --establish connection to the database
  sqlite.open;
  --execute sql statement
  sqlite.exec(sql);
  --close database
  sqlite.close;
end;

```

The delivery from the store should take place cyclically. For this purpose, a sequence\_position is increased by one and the corresponding part in the sequence is read. Thereafter, the name of the part in the database is searched. The first hit is read; the part will be moved from the store and the record is deleted from the database. Method delivery:

```

is
  id:integer;
  partName:string;
  part:object;
  sql:string;
do
  partName:=sequence.read(sequence_position);
  -- search part in the database
  sql:="SELECT * FROM inventory WHERE name='"+
  partName+"' LIMIT 1";
  --connect with database
  sqlite.open;
  -- prepare sql-statement
  sqlite.prepare(sql);
  --execute
  if sqlite.step then
    -- found
    id:=sqlite.getColumnInteger(1);
    part:=str_to_obj(sqlite.getColumnString(4));
  end;
  -- release memory
  sqlite.finalize;
  -- delete record
  sql:="DELETE FROM inventory WHERE id="+
  to_str(id);
  sqlite.exec(sql);
  --close database
  sqlite.close;
  -- move part
  part.move(line);
  --increase position in the sequence
  sequence_position:=sequence_position+1;
  if sequence_position > sequence.dim then
    sequence_position:=1; -- start new
  end;
end;

```

In the init method, delete the data from the database table to remove the data from the previous simulation runs:

```

is
do

```

```

sqlite.open;
sqlite.exec("DELETE FROM inventory");
sqlite.close;
end;

```

### 14.6.3 SQL Functions in SQLite

SQL provides a number of functions to evaluate values within a query result (Table 14.8). The functions are always used in connection with a SELECT query.

**Table 14.7** SQL functions

SQL function	Description
MIN (column)	Returns the minimum value in a column within a query result
MAX (column)	Returns the maximum value in a column within a query result
COUNT (column)	Number of values in a query result
AVG(column)	Average value
SUM/TOTAL(column)	Sum of the values in a column; TOTAL returns a real value, SUM the data type of the column (which can be a problem if the number is too large)

To query, for example, how many parts are included with the name Part1 in stock, you would have to formulate the following command:

SELECT COUNT(id) FROM inventory WHERE name='Part1'.

The result has only one value.

## 14.7 The XML Interface

You can use the XML interface to read and write data.

### 14.7.1 Introduction in XML

XML means Extensible Markup Language. XML is a mark-up language that specifies neither the available tags, nor the grammar of the target language. Each well-formed XML document must have a root element. A root element encloses all the other elements of an XML file. This means that all other elements are child elements of the root element; all these other elements must be within the start tag and the end of the root element tag.

Example: Customer data is to be stored in an XML file. There is one root element used: <customer\_base>. All customers (and the information about customers) are child elements of <customer\_base>. A corresponding XML file could have the following structure:

```
<?xml version="1.0" standalone="yes"?>
<customer_base>
  <customer>
    <lastname>
      Mueller
    </lastname>
    <firstname>
      Heinz
    </firstname>
    <customer_number>
      12345
    </customer_number>
    <address>
      <street>
        woodstreat
      </street>
      <street_no>
        48
      </street_no>
      <zipcode>
        09999
      </zipcode>
      <city>
        Torfhausen
      </city>
    </address>
  </customer >
</customer_base>
```

XML elements consist of a start tag, an end tag and the content. "<" and ">" must not be used for any other purpose. Element names must begin with a letter or an underscore. The rest of the name can consist of letters, numbers, underscores, hyphens and dots. Spaces and colons must not appear in the names. In addition, all names are prohibited that start with `xml` or `XML`, as these letter strings are reserved for future XML names. XML elements can have attributes like HTML elements. These consist of the name of the attribute and the attribute value that is written in double quotes (`name = "value"`). Therefore, for the example above, there is an alternative syntax:

```
<address street="Holzweg" street_no="48" zipcode="09999" city="Torfhausen"/>
```

Attribute names can appear only once; hence, it is, e.g. impossible with the above form to set up a secondary residence. If an element can appear more than once, it must be set up as a child element.

### 14.7.2 Read in XML Files into Tables

You can read XML files with a built-in method of the table objects and store Plant Simulation table contents into XML files with relative ease. Plant Simulation loads the data from the XML files in multi-dimensional tables. This function helps, e.g. to develop a quick understanding of unknown XML files.

#### Example: Read in XML into a table

First, load the example of `xml_interface` from [www.bangsow.de](http://www.bangsow.de). You find in the zip file an XML file (`example_en.xml`). Unzip this file into the directory where your Plant Simulation file is located. Insert a frame with a table (name: `tableFile`) and a method. You can read XML files with the command `<path>.readXMLFile(<string>)`. To read the file `example_en.xml` into the table, use the following method:

```
is
do
    tableFile.readXMLFile("example_en.xml");
end;
```

Plant Simulation inserts the name of the node as a column heading in the table. The root node of the file is called "coordinates"; below it, the nodes are labeled with "item" and each "item" has the following properties: id, card, location, x and y. In the XML file, coordinates on a card are stored (Fig. 14.28).

Fig. 14.28 Imported XML file

### 14.7.3 The XML Interface

Plant Simulation provides the XML interface for reading and writing access to XML files. A distinction is made between sequential and random access. In sequential access, you search in the XML file (usually with recursive methods) until you find what you are seeking. For random access, you address the node you want to read and write. There is no sequential searching of the XML file. Addressing the node is done using XPath statements. The following chapter deals with random accesses using XPath statements.

### Integrating an XML File into the Model

To embed an XML file, you must add an object of type XMLInterface to your network and specify the file you want to open in the interface. The file will be fully loaded into memory.

#### Example: XML Interface

First, load the example of xml\_interface from [www.bangsow.de](http://www.bangsow.de). You find in the zip file an XML file (example\_en.xml). Unzip this file into the directory where your Plant Simulation file is located. Insert in a frame a table, a method and a XMLInterface. Open the XML interface and select the file example\_en.xml (Fig. 14.29).

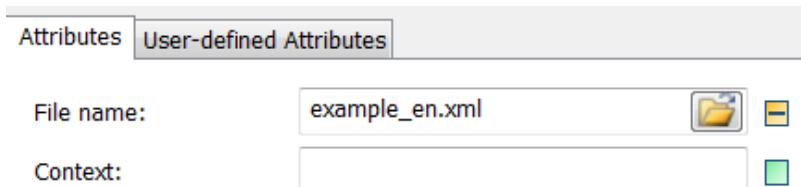


Fig. 14.29 XML interface

#### 14.7.3.1 Select Nodes and Read Values

To read the attributes of a node, follow the sequence below:

1. Open the file
2. You "get" the node that interests you; the information of the node will be written into a table
3. Close the file
4. Read the data of your interest from the table and make them available to the simulation

You need the SimTalk methods of the XML interface (Table 14.9):

Table 14.8 SimTalk methods of the XML interface

Method/Attribute	Description
<path>.openDocument	Opens the XML document
<path>.selectNodes(<string>)	Using selectNodes, you can select one or more nodes; you must pass an XPath address
<path>.getNodeValue(<string>)	Reads the value of the selected node
<path>.getNodes(<string>, : table	<integer>)GetNodes reads <integer> layers from the node specified by <string> and returns a table with the values; formulate the path of the node in XPath; the integer value determines the number of levels that are to be read—zero means that no child elements will be read
<path>.close	Closes the XML file

The XPath addresses are constructed similarly to addresses in a UNIX operating system. There are signs for the root (""). Nodes within the address are separated by slashes. You can also first select a node and then move relative to the node (relative paths). These addresses do not begin with a slash.

XPath provides the following expressions:

**Table 14.9** XPath expressions

XPath expression	Description
Nodename	Selects all child elements of the node named "Nodename"
/	Selects all nodes from the root node
//	Selects all nodes, regardless of their position
.	Selects the currently selected node
..	Selects the parent node of the selected node
@name	Selects the attribute (name) of the selected node

#### Example: XML Interface (continuation)

First, test some XPath statements. Enter the following structure in the method. In the following, you will only change the XPath statements.

```
is
tbl:table;
do
    XMLInterface.openDocument;
    -- select node with XPath
    tbl := XMLInterface.getNodes("/coordinates", 2);
    XMLInterface.close;
    tbl.openDialog(true);
end;
```

Set a breakpoint on the line class `tbl.openDialog` so that the table remains open. The tables have a uniform scheme (Fig. 14.30).

	string 1	string 2	string 3	table 4	table 5
string	Name	Namespace	Value	Attributes	Descendants
1	coordinates				Descendants_1

**Fig. 14.30** XML data format

Use the table `Descendants` to reach the next level, if you have set it in the command `getNodes` (in the case above 2). Now change the path to `"/coordinates/item"` and start the method again. The table now contains only the item nodes. If you change the path to `"/coordinates/item/location"` now, the table will contain the location value of each item node.

You can restrict the selection of nodes using the predicates. Predicates are written in square brackets. With predicates, you can search for specific values in the XML file, or filter for specific values. The following predicates are available:

**Table 14.10** XPath predicates

XPath predicate	Description
<code>/node/child[1]</code>	Selects the first child element of a node
<code>/node/child[last()-1]</code>	The second-last child element
<code>/node/child[position()&lt;x]</code>	Selects child elements up to a certain position
<code>/node/child[value='xxx']</code>	Selects the child elements that have a certain value in the value element; you can use <code>&gt;&lt;</code> when there are numeric values; these comparisons can also occur within the path

**Example: XML interface (continuation)**

Check the following XPath expressions:

Expression	Description
<code>/coordinates/item[1]</code>	Only the item element
<code>/coordinates /item[1]/location</code>	Only the location of the first item element
<code>/coordinates/item[last()]/location</code>	Location of the last item
<code>/coordinates/item[position()&lt;5]/location</code>	Locations of the first four items
<code>/coordinates/item[card=3]/location</code>	Locations of all items that have the value three in the child attribute map

You can use the pipe symbol ("|") to combine path components. Thereby, you can create excerpts from larger elements. For example, indicate the name and the X and Y values of the first five elements in our example—the XPath statement would look like this:

```
/coordinates/item[position()<6]/location|/coordinates/item[position()<6]/x|/coordinates/item[position()<6]/y
```

The predicates can be linked to more complex expressions with a number of operators like “and/or” or “!=” for not equal.

**Example: XML interface (continuation)**

All item elements (location) should be displayed that belong to the card 3 and have an x-value of > 400: The XPath statement must be:

```
/coordinates/item[card=3 and x>400]/location
```

If the data have unique identifiers (e.g. if the XML file was created by exporting from a database), you can select the node in question on the ID and read the data from the node individually. For this, you need more SimTalk commands. The result of the XPath statement is first summarized in a node. Again, you need to dissolve this node to get to the actual data. The dissolution of a node (moving one level down) is done with `<path>.selectChildren`. One level up, you can move with the command `<path>.closeChildren`. Using `<path>.getNextNode`, you can select the child elements and read the data.

**Example: XML interface (continuation)**

Add a second method in the frame. The readout of the data of a particular coordinate node might look like this (you want to read the location, x and y coordinate of the item with ID = 3).

```

is
  location:string;
  x:string;
  y:string;
do
  XMLInterface.openDocument;
  -- select node via XPath
  XMLInterface.selectNodes("/coordinates/item[id=3]"+
    "/location | /coordinates/item[id=3]/x"+
    " | /coordinates/item[id=3]/y");
  -- select all nodes
  XMLInterface.selectChildren;
  -- read one value after the other
  XMLInterface.getNextNode;
  location:= XMLInterface.getNodeValue;
  XMLInterface.getNextNode;
  x:= XMLInterface.getNodeValue;
  XMLInterface.getNextNode;
  y:= XMLInterface.getNodeValue;
  -- back to the parent node
  XMLInterface.closeChildren;
  XMLInterface.close;
  print location+" ("+x+", "+y+" ) ";
end;

```

**14.7.3.2 Changing Data in XML Files**

You can change a node's data by selecting the node, load data into a table, change the data in the table and then write the modified data back to the node of the XML file. You will need two additional Plant Simulation methods (Table 14.11).

**Table 14.11** SimTalk methods for writing in XML files

Method	Description
<path>.updateNodes(<table>)	Updates the data of the selected node corresponding to the data in the transferred table
<path>.write	Overrides the XML file with the updated data from memory

**Example: XML Interface (continuation)**

The x-value of the entry with ID = 3 should be set to 500. Add a new method to the frame. The following commands are necessary for this:

```
is
  tbl:table;
do
  XMLInterface.openDocument;
  -- select node for changing
  tbl := XMLInterface.getNodes(
    "/coordinates/item[id=3]",1);
  -- change value in the table, Position [3,1]
  --tbl.openDialog(true); -- for checking
  tbl[5,1][3,4]:="500";
  --update value in the memory
  XMLInterface.updateNodes(tbl);
  -- update file
  XMLInterface.write;
  XMLInterface.close;
end;
```

# Subject Index

? 25  
@ 25  
3D animation path 582  
3D extrusion 586  
3D graphic 580  
3D layout 578  
3D MU animation 586  
3D Properties 581  
3D self-animation 590  
3D state icon 584  
3D unnamed animation 591  
3D viewer 580

## A

abs 47  
absolute path 25  
AbsSimTime 212  
Abstraction 5  
acceleration 28, 310  
Accumulating 283  
ActiveX 672  
ActiveX Data Objects 677  
ADO 677  
ADOX 677  
Analysis 5  
AND 33  
AngularConverter 286  
animated camera 592  
animation events 570  
animation line 566  
animation point 566  
anonymous identifier 24  
any 28  
append 184  
array 29  
assembly 162  
assemblyList 165

AssemblyStation 162  
assignment 33  
AttributeExplorer 209  
automaticSetup 126  
Availability 4, 108  
AvailableForMediation 596  
average inventory 439  
AVG 697

## B

background 563  
Backwards 284, 304, 310  
batBasicCons 351  
batCapacity 351  
batch processing 149  
batCharge 351  
batChargeCurrent 351  
batCharging 351  
batDriveCons 351  
batReserve 351  
battery 350  
beep 100  
before actions 48  
bell 100  
Bill of Material 164, 231  
bin Kanban system 239  
blocking 106  
BOF 681  
bol\_to\_num 35  
BOM 164, 231  
bookmark 19  
boolean 28  
BottleneckAnalyzer 644  
branch 35  
breakpoint 61  
Broker 499  
BrokerPath 528

Buffer 411, 685  
byref 43

## C

CAD drawing 565  
calculateWorkingDuration 112  
calendarWeek 212  
callback argument 82  
callback function 82  
camera path animation 593  
Capacity 4, 106, 132, 284, 621  
CardFile 182  
Carry Part Away 507  
case differentiation 37  
ceil 47  
cEmp 263  
chain conveyor 283  
Chaku-Chaku 514  
channel 661  
chaotic warehousing 417  
Chart 633  
checkbox 88  
childNo 609  
chromosome 281  
class 55, 424, 608  
Class breakpoint 64  
class library 9  
close 94, 694  
closeDialog 94  
CloseHTMLWindow 101  
code completion 20  
collided 310  
COM interface 667  
comment 26, 649  
conditional loop 38  
connect 74  
Connection 678  
connection string 678  
Connector 10, 76  
console 9  
constant work in progress 246  
constructor 52  
cont 132, 136, 304  
control loop 229  
converter 292  
CONWIP 246  
copy 45, 185  
COUNT 697

create 133, 310, 536  
CREATE TABLE 692  
createNestedList 191  
createObject 74  
createSensor 301  
crossroads 333  
cross-sliding car 337  
cumulative quantities 445  
Current 25  
currentAmount 547  
currentFillAmount 557  
currentFlowRate 559  
currIcon 573  
currIconNo 573, 576  
cursor 184  
cursorX 186  
cursorY 186, 422  
Curve 286  
cutRow 184, 422

## D

data collection 261  
data type 27  
DataFit 267  
date 28  
datetime 28  
day 212  
dayOfWeek 212  
DDE 661  
DDE Hotlink 666  
DDEExecute 662  
DDEInitiate 661  
DDEPoke 662  
DDERequest 662  
DDETernate 662  
debugger 22  
declaration 27  
delAniPoints 573  
delete 134, 184, 185  
DELETE 689  
deleteMovables 132  
delivery table 139  
delivery time 437  
dEmp 263  
Derivation 13  
destination 310, 510  
destination list 319  
destructor 52

dialog 79  
dim 184, 186  
DismantleStation 169  
dismantling operations 169  
DismantlingStation 305  
display 645  
display panel 647  
do 18  
downto 40  
drag and drop 53  
Drag-and-Drop 302  
drain 138  
drop-down list box 89  
Duplication 13  
dynamic data exchange 661  
dynamic model generation 73

**E**

Economic Order Quantity 444  
efficiency 524  
e-Kanban 229  
elseif 36  
Emp 263  
empirical distributions 261  
eMPlantLib 667  
empty 120, 184, 548  
end tag 698  
endless loops 38  
endSequence 353  
EndSim 23  
energy consumption 599  
energy state 599  
EnergyAnalyzer 599  
engage 531  
entry gates 105  
EOF 681, 682  
ErrorHandler 64  
EventController 211  
exec 694  
Execute 681  
ExitLoop 40  
exitStrategy 528  
Exporter 499

**F**

failed 107, 120, 637  
failure importer 500

Fields 681  
file interface 682  
FileName 682  
finalize 694  
find 184, 186, 422  
fireSimTalkMessage 671  
fitness value 280  
fixed transfer line 159  
flexible cycle lines 160  
floor 47  
flow rate 543  
FlowControl 176  
FluidDrain 543  
Fluids library 543  
FluidSource 543  
footer controlled loops 39  
footpath 502  
forklift 357  
for-loop 39  
frame 69  
free 488  
from-loop 39  
front 328  
frontMU 342  
frontPos 318  
full 132, 420  
Function 41

**G**

GA 277  
GA range allocation 279  
GA sequence optimization 277  
Gantry 405  
GAWizard 279  
generator 206  
generic algorithms 277  
getCheckBox 88  
getColumnCount 694  
getColumnInteger 694  
getColumnString 694  
getCreationTable 533  
getCurrShift 112, 271  
getDate 212  
getFrontWagon 342  
getIcon 85  
getIconSize 573  
getNodes 700  
getNodeValue 700

getObjects 56  
 getPart 488  
 getPartFromPosition 488  
 getPartFromPositionToObject 488  
 getPartToObject 488  
 getPixel 573  
 getPortals 353  
 getRearWagon 342  
 getStoreTable 489  
 getStoreXDim 489  
 getStoreYDim 489  
 getStoreZDim 489  
 getTable 93  
 getTableRow 93  
 getTimesTable 377  
 getTractor 342  
 getValue 83, 668, 670  
 global variables 30, 31  
 goTo 511  
 GoToLine 682  
 goToPool 511  
 graphic structure 595  
 gravity-roller conveyor 283  
 group box 93  
 GroupID 87

**H**

header controlled loops 38  
 histogram 640  
 hitchFront 342  
 hitchRear 342  
 hook 353  
 HTMLReport 658

**I**

icon 563  
 Icon Animation 120  
 icon editor 563  
 idle time 629  
 if 35, 36  
 Importer 499  
 importerActive 528  
 incl 45  
 InflowRate 553  
 infobox 100  
 Init 23  
 initialize 185

InitStat 617  
 Insert 184, 688  
 insertRow 186  
 inspect 37  
 instance 12  
 integer 28  
 integer division 32  
 interface 69  
 internalClassName 55  
 is 18  
 IsRunning 212, 216  
 isSetupFor 126  
 isTractor 342  
 Item 95, 661, 681

**J**

jobs per hour 622  
 JPH 622  
 JT format 580

**K**

Kanban 228  
 Kanban card 229  
 KanbanBuffer 244  
 KanbanChart 246  
 KanbanSingleProc 244  
 KanbanSource 244

**L**

LED 120  
 Length 28, 284  
 library 72  
 light barrier mode 299  
 line 283  
 list 28  
 list box 89  
 list editor 181  
 ListView 90  
 load bay 304  
 LoadBayBackwards 337  
 LoadBayLength 337  
 LoadBaySpeed 337  
 loadingTime 514  
 LoadModel 668, 670  
 local variable 27  
 location 519

LockoutZone 403  
logical errors 62  
login 686  
logout 686  
loop 38

**M**

machine-hour rate 610  
main camera 592  
makeArray 30  
makeRGBValue 482  
material consumption 436  
material costs 607  
MaterialsTable 543  
Mathematical operators 32  
Max 47, 190, 697  
maxAttr 190  
maxXDim 200  
mean time between failures 110  
mean time to repair 108  
meanValue 190, 621  
meanValueAttr 190  
menu item 93  
messagebox 98  
methCall 54  
Method 17  
method call 47  
MethWakeu 58  
Min 47, 190, 697  
minAttr 190  
Mixer 543  
model 2  
modulo 32  
money 28  
movable units 283  
move 134  
moveHook 353  
moveTo 353  
moveToObject 353  
moveToPosition 353  
movingSpeed 310  
MTBF 110  
MTTR 108  
mu 132  
multi machine operation 509  
multi-container system 456  
Multi-level experimental design 276  
multiple failures 110

multiple parts processing 157  
multiple-machine operation 527  
multi-portal crane 353  
muPart 132  
MUTarget 514, 528

**N**

name scope 25  
nested tables 191  
node 55, 608  
NOT 33  
num\_to\_bool 35  
numChildren 609  
numMu 132  
numNodes 55  
numOfLimitedObjects 14  
numSensors 328  
numSucc 324

**O**

object 28  
observable value 60  
observer 57  
occupied 132, 524  
ODBC interface 684  
off 599  
omit 45, 664  
open 94, 694  
openColorSelectBox 96  
openDialog 94, 701  
openDocument 700  
OpenHTMLWindow 100  
openObjectSelectBox 96  
operating state 599  
operational 120  
operators 32  
OR 33  
order management 221  
order rhythm method 447  
outflowrate 550  
overall system availability 250

**P**

ParallelProc 154  
Passing arguments 41  
path 25

pause 120  
 paused 107  
 pe 44, 136, 420  
 pearl chain 254  
 PickAndPlace 363  
 pick-up principle 228  
 Pipe 543  
 pipeOpened 546  
 PlaceBuffer 411  
 placeIsFree 489  
 PLM Software Community 7  
 plotter 633  
 portal loader 405  
 Portioner 543  
 pos 45  
 position 328  
 position-time diagram 318  
 positionType 328  
 pow 47  
 Power up early 600  
 powerInputOperational 605  
 PowerInputWorking 602  
 prepare 694  
 previousLocation 520  
 prio 59  
 probability distributions 261  
 processing time 106  
 procurement 446  
 production control 216, 225  
 production costs 606  
 production Kanban 229  
 production planning and control 216  
 production sequence 254  
 profiler 66  
 program 657  
 Prompt 41, 96  
 promptList1 96  
 promptList1N 96  
 pull control 227  
 putValuesIntoTable 636

## Q

quality control 176  
 queue 28  
 QueueFile 183  
 quit 668, 670

## R

radio button 87  
 read 182, 184  
 ReadLn 682  
 readXMLFile 699  
 ready 129  
 real 28  
 rear 328  
 rearMU 342  
 recipe 543  
 Recordset 680  
 recovery time 105  
 recursion 55  
 ref 55, 657  
 reference point 566  
 relative path 25  
 RemainingProcTime 124  
 remainingSetupTime 126  
 RemoteControl 667  
 Remove 682  
 reorder level 435  
 Reorder point 435, 455  
 reorder point method 456  
 Repeat 39, 98  
 Replenishment lead time 435  
 report 650  
 report header 651  
 resBlocked 128  
 reserve 489  
 Reset 23, 212  
 ResetSimulation 670  
 resetStatement 694  
 resource management 224  
 resourceType 55  
 ResStatOn 616  
 result 43  
 resWaiting 128  
 resWorking 128, 514  
 return 43  
 rework 176  
 Root 24  
 Rotation Path 586  
 round 47  
 route weighting 320  
 routing 319  
 runtime errors 61

**S**

safety stock 435  
Sankey diagram 641  
Scaling Factor 74, 285, 565  
scalingFactor 301  
schedule 112, 212  
scrap rate 176  
segmentsTable 77, 301  
SELECT 690  
selectFileForOpen 96, 673  
selectFileForSave 96, 677  
selectNodes 700  
Self 24  
sensor 47, 296  
sensorID 301  
sensorNo 328  
sequence stability 256  
setAniPoint 573  
setBackgroundColorCells 482  
setCaption 83, 87  
setCheckBox 88  
setCreationTable 534  
setCurrentContent 546  
setCursor 186, 422  
setDestination 289, 376  
setGraphicMaterial 596  
setIcon 85, 122  
setIconSize 573  
setIndex 90  
setPixel 573  
setSensitive 83, 87, 93  
setServices 517  
setTable 93  
setTableRow 93  
setTimesTable 377  
setup 126  
setup time 105  
setupFor 126  
setupTime 126  
setValue 670  
ShiftCalendar 111  
shiftCalendarObject 112  
shiftPart 489  
shiftPlan 112  
SimTalk 17  
SimTime 212  
Simulation 2

simulation run 2  
single processing 148  
sort 186  
sorter 412  
source 138  
Speed 28, 284, 304, 310, 524  
speed-time graph 313  
splitString 45  
sql 686  
SQLite 691  
stack 28  
StackFile 183  
standby 599  
start 212  
Start delay duration 333  
start tag 698  
startPause 310  
startPauseIn 310  
startSimulation 670  
startStat 212  
statAvgExitInterval 624  
statAvgLifeSpan 624  
statBatChargePortion 627  
statBlockingPortion 617, 637  
state 353  
state icons 122  
StatEmptyPortion 617  
statExporterFailedPortion 501  
statExporterOperationalPortion 501  
statExporterPausedPortion 501  
statExporterUnplannedPortion 501  
statFailPortion 617, 637  
static text box 83  
statistical data 615  
Statistics 615, 617  
statistics collection period 615  
Statistics Wizard 639  
StatMaxNumMU 617  
statMediationTime 501  
statMediationTimeMU 501  
statNumIn 251, 617, 622  
StatNumOut 617  
statOpenRequests 501  
statPausingPortion 617, 637  
statRelativeOccupation 439, 621  
statSatisfiedRequests 501  
statServicesFailedPortion 501  
statServicesRepairingPortion 501

statServicesSetupPortion 501  
 statServicesWaitingPortion 501  
 statServicesWorkingPortion 501  
 StatSetupPortion 617  
 statStayTime 501  
 statStayTimeMU 501  
 statStoreWaitingPortion 627, 629  
 statSumFreeCapacity 501  
 statThroughputPerHour 624  
 statTransportTimePortion 627  
 statTranspWaitingPortion 627  
 statTranspWorkingPortion 627  
 statTspBlockingPortion 627  
 statTspFailPortion 627  
 statTspPausingPortion 627  
 StatUnplannedPortion 617  
 statWaitingPortion 617, 637  
 StatWorkingPortion 617, 637  
 step 212, 694  
 Step-by-Step Help 6  
 stop 212  
 stopped 310  
 stopuntil 59  
 StorageCrane 486  
 store 417  
 storePart 489  
 storePartFromObject 489  
 str\_to\_bool 35  
 str\_to\_date 35  
 str\_to\_datetime 35  
 str\_to\_length 35  
 str\_to\_num 35  
 str\_to\_obj 35  
 str\_to\_speed 35  
 str\_to\_time 35  
 str\_to\_weight 35  
 string 28  
 strlen 45  
 sub-table 191  
 succ 324  
 Sum 190, 697  
 sumAttr 190  
 supermarket principle 228  
 syntax errors 61  
 sysStart 216  
 system 2

## T

tab control 93  
 table 28

TableFile 181  
 Tank 543  
 target control 375  
 template 22  
 testExportFor 531  
 textured plate 589  
 theme 661  
 ThroughputPerDay 624  
 time 28  
 timeOfDay 212  
 timeout 53  
 TimeSequence 28, 193  
 to\_str 35  
 toolbox 9  
 TOTAL 697  
 track 295  
 tractor 342  
 transaction 661  
 transfer 134  
 TransferStation 307  
 transition time 600  
 transport Kanban 229  
 transport process 283  
 transporter 295  
 trigger 143, 197  
 trim 45  
 Turnplate 290  
 turntable 288  
 Tutorial 6  
 two-dimensional array 30  
 TwoLaneTrack 334  
 type conversion 34  
 typeStatistics 624  
 typeStatisticsCumulated 624

## U

unhitchFront 342  
 unhitchRear 342  
 uniform distribution 270  
 unloadingTime 514  
 until 39  
 UPDATE 688  
 updateNodes 703

## V

value 95  
 variable 27  
 VBA 661

Visual Basic for Applications 661

VRML 580

## W

wait 58

waituntil 59

warehouse 411

warehouse costs 443

warehouse model 461

warehousing 416

warehousing strategies 447

warm-up time 271

week 212

weight 28

when 37

WHERE 689

while 38

work plan 217, 219

Worker 4, 501

WorkerChart 509

WorkerPool 502

working assets 608

workstation 502

write 703

WriteExcelFile 618

WriteFile 618

WriteLn 682

writeRow 186, 421

## X

xDim 185, 310, 420

XML interface 697

XPath 701

xPos 77

## Y

yDim 185, 310, 420

year 212

yPos 77

## Z

z\_dEmp 231

zoomX 573

zoomY 573