IBM ILOG Plant PowerOps V3.2

Documentation

Table of contents

Getting Assistance......11

Contacting IBM Support......12

How to use the documentation......15

Publications in the documentation set......16

The documentation formats......17

Accessing the CHM, PDF, and HTML documentation......18

Disclaimers......19

Acknowledgement......20

System requirements......21

IBM ILOG Plant PowerOps V3.2 Release Notes......23

System requirements......25

User Interface improvements......27

Model checker and data reader......28

Data exchange with Microsoft Excel......29

Improved navigation and filters......30

Editing data in the GUI......31

New design and function for some views and tools......33

Modeling changes......35

Promised demand used in interactive planning......36

Modeling initial stock......37

Persistence of planning decisions......38

Enhancements to planning, batching, and scheduling...39
Planning module enhancements...41
    Dynamic safety stock...42
    Infeasibility management..43
Batching algorithms..45
Scheduling...46
Relational model changes...47
Separation of master data and transactional data...48
Migration to schema version 4.0..49
Data schema modeling changes...50
Reporting with Tableau..54
Documentation, acknowledgement..55
Publications in the documentation set..56
Accessing the CHM, PDF, and HTML documentation...57
Acknowledgement...59
Implementation of Plant PowerOps...61
Introduction to planning and scheduling with PPO...63
Overview..64
What Is IBM ILOG Plant PowerOps?...65
How Plant PowerOps solves the problem..67
Optimization and KPIs..69
Optimization modules and solution..70
Key Performance Indicators...73
Optimization profile..76
Advanced optimization...79
    Production planning advanced options...80
    Detailed scheduling advanced options..82
    Constraints..83
    Post processing...84
    Parameters for optimization...86
    Complexity considerations...89
    Problem decomposition..90
The basic data model..91
Overview of data modeling for PPO..93
    Introduction to a data model, master data, and transactional data...94
    Costs and revenue...96
    Methods of modeling data..97
Global model information...99
    The model environment...100
    Time units and time buckets..101
    Planning horizon, scheduling horizon, and end max...102

Materials....103
Resources....105
Demands and due dates....109
Recipes....111
Production orders and material flow arcs....115
Procurements....117
Firm or fixed information....118

Extended use of the data model....119

Production planning considerations....121
- Inference of the planning problem from the scheduling problem....122
- Defining time buckets....123
- Recipe validity periods....124
- Formula optimization and blending....125

Resources: Extended usage....127
- Planning with super resources....128
- Setup times and costs....130
- Cleanups....132
- Resource validity periods....133

Calendars....135
- Why use calendars?....136
- Calendar intervals....137
- Overlapping calendar intervals....138
- Calendars on modes or resources....139
- Breaks and shifts....140
- Productivity....141

Activity constraints, validity periods, and names....143
- Precedence constraints....144
- Spanning constraints....145
- Compatibility constraints....146
- Activity validity periods....147
- Activity names....148

Material lifespan and inventory....149
- Maturity, shelf life, waste recipes....150
- Managing inventory stock....153

Applied use of Plant PowerOps....161

Using the PPO GUI....163

GUI tool and navigation tips....165
- Menus and toolbars....167
- Accessing the plan views....179
- Splitting and synchronizing plan views....181
- Using the Filter tool on the Master and Transactional Data tables....183
- Comparing multiple scenarios....184

Repair extent, capacity and magnetism.....185
Stock Coverage view.....186
The Inspector.....188
The Parameters window.....190
Using the checker.....191
Copying an existing recipe.....193

Database usage and connectivity.....195

Supported databases.....196
Basic database connectivity.....197
Opening a database in PPO.....198
Automatic bucket generation.....202
Save scenario in a database.....205
Adding custom database connectivity.....206

Using service levels, lead time, and demand variability to manage stock levels.....209

Manually defining stock levels and corridors.....210
Service level concepts.....211
Uncertain demand and forecast error.....212
Production lead time.....213
Service levels.....215
Examples of using service levels, demand variability, and production lead time variability.....221
Service level in the GUI.....222
Using the cycle service level.....224
Using the fill rate service level.....228
Dynamic service level example.....229
Uncertain production lead time example.....231

Decomposition framework.....233

Introduction to decomposition.....234
Building the decomposition framework.....235
Optimization using scopes in the GUI.....240
Advanced usage of the decomposition framework using Java API.....241

Production planning simulations.....243

Interactive planning.....244

Advanced usage: Distribution planning.....247

General concepts.....248
The distribution plan calculation.....250
Modeling the distribution plan.....251
Key modeling tables.....252
Building a distribution plan model.....253
Viewing the results.....263

Examples and tutorials.....267

Modeling a simple problem: A “bottleneck” resource.....269

Overview....271
Describe the problem....273
    Overview....274
    Objective and costs....275
    Before creating the model....277
An overview of the PPO data model....279
    Model the process....281
    Materials....282
    Demands....283
    Activities....284
    Resources....285
    Recipes....286
    Modes....287
    Material production....288
    Production orders....289
    Production to demand arcs....290
    Weighted objectives....291
    Overview of the process....292
    Table of data model objects....293
A quick tutorial on csv and mdb usage in PPO....296
Model the problem....298
Solving....309
    Overview....310
    Solve in the PPO GUI....311
View and study the plan....312
Review....317

Using the PPO API for C++ to model and solve....319

Solving using csv files with C++....320
Model and solve using the API for C++....323
    Overview....324
    General approach to C++ modeling for Plant PowerOps....325
    Define the necessary C++ functions....326
    Build the C++ program....329

Using the PPO API for Java to model and solve....345

Solving using csv files with Java....346
Model and solve using the API for Java....347
    Differences between C++ and Java....348
    Define the program Java functions....350
    Build the Java program....353

Modeling a dairy plant with PPO Java API....367

Overview....368
The fresh dairy plant and its yogurt process....369
Building the overall model structure....373

Overview....375
Setting up the model....376
Defining the time buckets....377
Defining semi-finished intermediate products....378
Defining finished products....379
Days of supply....380
Resources for finished products....381
Defining the units of measure....382
Modeling a warehouse as a storage unit....383
Defining the production recipes for the finished products....384
Defining cleanup recipes....386
Defining initial stock....388
Defining the demand....389
Dealing with setups and obtaining the preferred sequence....390
Defining the costs and the weights....393
Modeling the manufacturing process....395
Representing process equipment....396
How to represent equipment connections....398
Modeling the intermediate materials and process activities....399
Internal constraints of the recipe....401
Tanks as storage units....403
Production, consumption and destination....404
Cleaning policy....405
Calendars and breaks....406
Reference Documentation....407
Customizing and Extending PPO....409
Authentication and access rights....411
User role determines access....412
Adding a login panel to PPO....413
Advanced configuration of roles and access rights....415
General security considerations....416
Customizing report generation with Tableau....417
GUI extension mechanism....419
Overview....420
Format of plug-in manifest files - plugin.xml....421
Extensions and extension points....424
Specific customizations....425
Overview of extension points....427
Plan view customization....431
Overview....432
PlanViewTypes extension point....433
PlanViewContainer.Configuration extension point....441
PlanViewContainer.Defaults extension point....448

Plan view extension example....451
Customizing views, menus, and toolbars....455
Adding new table layouts....456
Inserting a new panel view....457
Insert a new panel and toolbar in an existing view....463
Insert a new item in the menu and a new icon button in the main toolbar....465
Remove menu item and button from the main toolbar....466
Engine optimizer extensions....467
Overview....468
Write the engine optimizer factory class....469
Write the engine optimizer class to solve data....473
Configuring the data views....479
Overview....480
Activate the customization in a plug-in....481
Add a new table....483
Remove a table....489
Modify a table....490
Database customization....491
Installing JDBC jars....493
Customizing PPO data model tables....497
Options of the PlantPowerOps executable file....504

Using PPO with Microsoft products....505

Overview....506
Build and run Plant PowerOps examples....507
Creating a project workspace and link the target with PPO....508

Entity Relationship Diagrams....511

General tables....513
Activities, modes, and recipes....514
Cleanup constraints....515
Demands....516
Manufacturing resources....517
Material flow....518
Materials and storage units....519
Procurements....520
Production orders....521
Production plans....522
Production schedules....523
Setup times and setup costs....524

Universal Modeling Language diagrams....525

Date and time display....529

Overview....530
C++ runtime date/time output....531

Overview....532
C++ API date/time methods....534
Java runtime date/time output....535
Overview....536
Java API date/time methods....539
Time zone settings....540
Index....557

Getting Assistance

This section provides information on customer support and IBM® ILOG® Plant PowerOps (PPO) documentation.

In this section

Contacting IBM Support
Contains information on how to obtain technical support from IBM worldwide, should you encounter any problems in using IBM products.

How to use the documentation
Describes Plant PowerOps (PPO) documentation.

System requirements
This section lists the system requirements.

Contacting IBM Support

IBM Software Support Handbook

This guide contains important information on the procedures and practices followed in the service and support of your IBM products. It does not replace the contractual terms and conditions under which you acquired specific IBM Products or Services. Please review it carefully. You may want to bookmark the site so you can refer back as required to the latest information. The "IBM Software Support Handbook" can be found on the web at *http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html*.

Accessing Software Support

When calling or submitting a problem to IBM Software Support about a particular service request, please have the following information ready:

- IBM Customer Number
- The machine type/model/serial number (for Subscription and Support calls)
- Company name
- Contact name
- Preferred means of contact (voice or email)
- Telephone number where you can be reached if request is voice
- Related product and version information
- Related operating system and database information
- Detailed description of the issue
- Severity of the issue in relationship to the impact of it affecting your business needs

Contact by Web

Open service requests is a tool to help clients find the right place to open any problem, hardware or software, in any country where IBM does business. This is the starting place when it is not evident where to go to open a service request.

Service Request (SR) tool offers Passport Advantage clients for distributed platforms online problem management to open, edit and track open and closed PMRs by customer number. Timesaving options: create new PMRs with prefilled demographic fields; describe problems yourself and choose severity; submit PMRs directly to correct support queue; attach troubleshooting files directly to PMR; receive alerts when IBM updates PMR; view reports on open and closed PMRs. You can find information about assistance for SR at *http://www.ibm.com/software/support/help-contactus.html*

System Service Request (SSR) tool is similar to Electronic Service request in providing online problem management capability for clients with support offerings in place on System i, System p, System z, TotalStorage products, Linux, Windows, Dynix/PTX, Retail, OS/2, Isogon, Candle on OS/390 and Consul z/OS legacy products.

IBMLink SoftwareXcel support contracts offer clients on the System z platform the IBMLink online problem management tool to open problem records and ask usage questions on System z software products. You can open, track, update, and close a defect or problem record; order corrective/preventive/toleration maintenance; search for known problems or technical support information; track applicable problem reports; receive alerts on high impact problems and fixes in error; and view planning information for new releases and preventive maintenance.

Contact by phone

If you have an active service contract maintenance agreement with IBM , or are covered by Program Services, you may contact customer support teams by telephone. For individual countries, please visit the Technical Support section of the *IBM Directory of worldwide contacts*.

How to use the documentation

Describes Plant PowerOps (PPO) documentation.

In this section

Publications in the documentation set
Describes the information delivered with Plant PowerOps.

The documentation formats
Explains the different formats in which the documentation is provided.

Accessing the CHM, PDF, and HTML documentation
Describes how to access each of the different documentation formats.

Disclaimers
Documentation disclaimers.

Acknowledgement
Use of algorithm for computing the inverse normal cumulative distribution function.

Publications in the documentation set

Information delivered with Plant PowerOps is organized as follows:

- The *Release Notes*

  Introduces the new functionality and features of the most recent release.

- *Implementation of Plant PowerOps*

  Describes the optimization process, data modeling, optimization criteria and Key Performance Indicators. Serves as a companion reference to the data schema and API manuals.

- *Applied use of Plant PowerOps*

  Shows how to use the PPO GUI and data model to handle industry concerns such as managing stock corridors and service levels, and planning using *available-to-promise* calculations. Also includes some information on basic use of the GUI.

- *Examples and tutorials*

  Examples of creating data models to represent manufacturing processes, ranging from the basic to complex. Choose from examples using the data schema, C++ API or Java API.

- *Reference Documentation*

  Reference information includes the following:

  - *Customizing and extending Plant PowerOps*

    Modifying your PPO installation through customization, configuration changes, and extension by plug-in.

  - *Plant PowerOps Entity Relationship Diagrams*

    Diagrams showing model entities.

  - *Data schema*

    The complete list of data schema tables and fields with concise descriptions.

  - *C++ Reference Manual*

    The C++ API reference for PPO.

  - *Java Reference Manual*

    The Java™ API reference for PPO.

There is additional information available within the PPO GUI on tooltips, messages, and descriptive help text in various windows. The GUI also has the *Online Help* which provides basic information about system requirements, the menu and toolbars of the application, and shortcut keys.

The documentation formats

Most PPO documentation is available in several formats:

- ♦ CHM (Microsoft® Compiled HTML)
- ♦ PDF (Adobe® Portable Document Format)
- ♦ HTML (Cross-platform compatible)

Features and search

The CHM format provides a quick and easy-to-use search tool that functions across the entire PPO documentation set. It is a very thorough search mechanism; you may prefer to put your search term in quotes to enforce a strict search.

PDF files are useful for annotation, search, and printing purposes. As with CHM, each search hit is highlighted. You can download the Adobe Reader product for free from:

http://www.adobe.com/products

Cross-platform HTML contains all PPO documentation. This format is used with standalone browsers on Windows® and other platforms. There is no built-in search function available on the HTML documentation.

Note that search results may vary slightly across the documentation formats. For example, if searching for only part of a file or path name, you may get more useful results by searching the PDF documentation.

Accessing the CHM, PDF, and HTML documentation

CHM and PDF

CHM is available from the PPO GUI by selecting **Ctrl + D**. You can also launch the documentation from the PPO menu bar: **Help > Documentation.**

You can also access the documentation without starting PPO. For example on Windows® XP select: **Start > All Programs > ILOG > IBM ILOG Plant PowerOps > Documentation Plant PowerOps**.

The CHM and PDF documentation is located within your PPO product installation directory (represented as <*PPOInstallDirectory*>). You can find the CHM files in the folder:

<*PPOInstallDirectory*>`\doc\chm`

The PDF documentation is in the folder:

<*PPOInstallDirectory*>`\doc\pdf`

The naming convention for these files is as follows:

- **ppodoc**: The core information of the documentation set. In CHM format, this document includes access to all content.
- **ppocplus**: The *Plant PowerOps C++ Reference Manual*.
- **ppojava**: The *Plant PowerOps Java Reference Manual*.
- **pposchema**: The *Plant PowerOps Data Schema*.

If a title does not exist in the folder, then that publication is not available in that format. For example, the *Plant PowerOps Java Reference Manual* is not available as a PDF file.

HTML documentation

The HTML documentation is available at <*PPOInstallDirectory*>`\doc\html\en-US\documentation.html`. The HTML documentation is cross-platform compatible and includes all available publications.

Disclaimers

Acknowledgement

IBM® ILOG® Plant PowerOps includes code based on Peter J. Acklam's algorithm for computing the inverse normal cumulative distribution function: http://home.online.no/~pjacklam/notes/invnorm.

System requirements

- CPU of 2 GHz or faster.
- Memory of at least 2 GB.
- Windows® XP or Windows Vista.
- A large monitor with a 16:9 aspect ratio.

Recommended system specification for the database server

Oracle® 10g or 11g; or Microsoft® SQL Server® 2000 or 2005.

IBM ILOG Plant PowerOps V3.2 Release Notes

IBM® ILOG® Plant PowerOps V3.2 is officially supported in English (American), Japanese and Chinese (Simplified).

In this section

System requirements
This section lists the system requirements.

User Interface improvements
Improved tools, increased functionality, and redesign of views.

Modeling changes
There are changes in modeling demand used for interactive planning, modeling initial stock, and enforcement of planning mode decisions.

Enhancements to planning, batching, and scheduling
Describes new functionality available in the optimization modules for Plant PowerOps (PPO) V3.2.

Relational model changes
This section introduces the Plant PowerOps (PPO) schema version 4.0. The data schema goes through a major upgrade for PPO V3.2 due to separation of master and transactional data.

Reporting with Tableau
Tableau version updated to 4.1.

Documentation, acknowledgement

For the 3.2 release, PPO documentation features improved organization and full search in CHM format.

System requirements

- ♦ CPU of 2 GHz or faster.
- ♦ Memory of at least 2 GB.
- ♦ Windows® XP or Windows Vista.
- ♦ A large monitor with a 16:9 aspect ratio.

Recommended system specification for the database server

Oracle® 10g or 11g; or Microsoft® SQL Server® 2000 or 2005.

User Interface improvements

Improved tools, increased functionality, and redesign of views.

In this section

Model checker and data reader
More rules, better checking.

Data exchange with Microsoft Excel
Improvements in exchange of data.

Improved navigation and filters
You can filter by recipe or resource.

Editing data in the GUI
Enhancements to editing production order data and master data directly in the PPO GUI.

New design and function for some views and tools
A summary of GUI redesign efforts.

Model checker and data reader

The model checker contains more rules to guide you while building your model.

The data reader enforces more checking, logged in the PPO console.

Data exchange with Microsoft Excel

When importing data from Microsoft® Excel® , a new scenario is created in PPO, which merges the model of the current scenario with the imported tables.

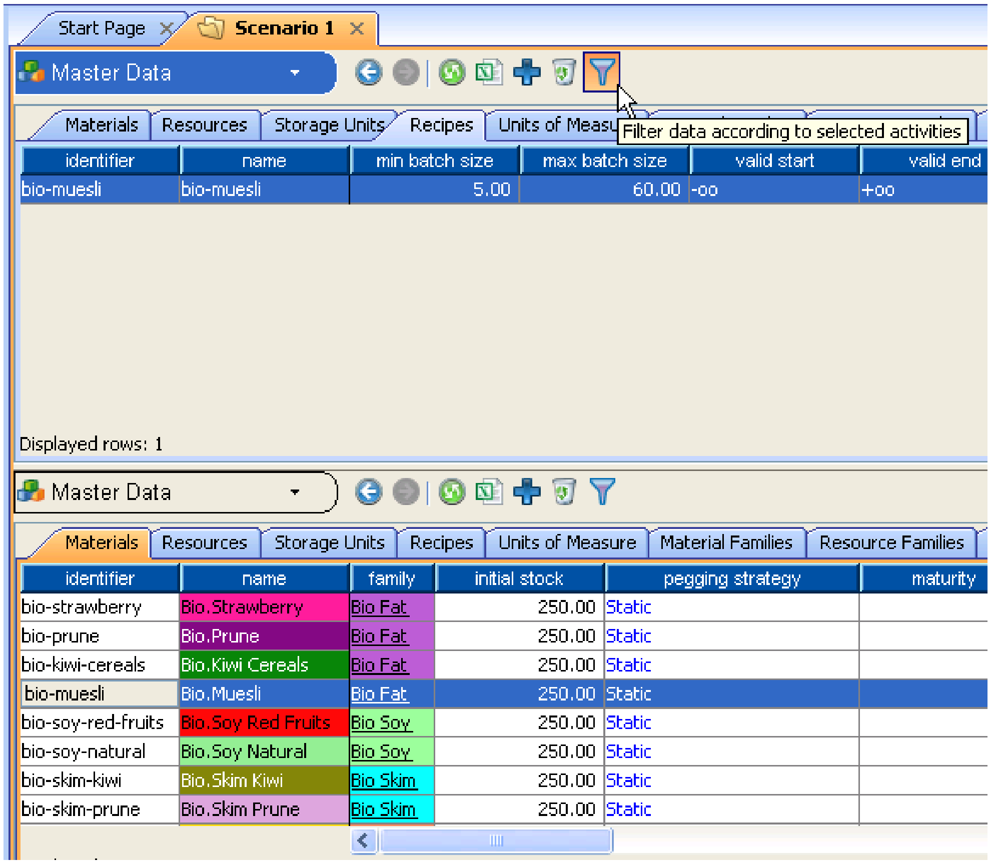
When exporting data to Excel, you can limit the number of exported tables. You can select the exact tables for export by individual table or by relational model group (such as recipe tables, production order tables, and so forth). Also, when exporting data, all dates are now real dates in the time zone of the model, instead of integer format.

Performance during import and export has greatly improved; re-importing spreadsheet data is up to 100 times faster than in previous versions.

Improved navigation and filters

When displaying two or more **Master Data** or **Transactional Data** tables, you can filter the displayed data by various criteria, including by materials, recipes and resources.

For example, the following image shows two **Master Data** views, the lower of which is focused on the **Materials** tab with the material **bio-muesli** selected. The upper view is focused on the **Recipes** tab, where the user had selected the filter icon with the result that the only recipe displayed is the one associated with the bio-muesli material.



identifier	name	min batch size	max batch size	valid start	valid end
bio-muesli	bio-muesli	5.00	60.00	-oo	+oo

Displayed rows: 1

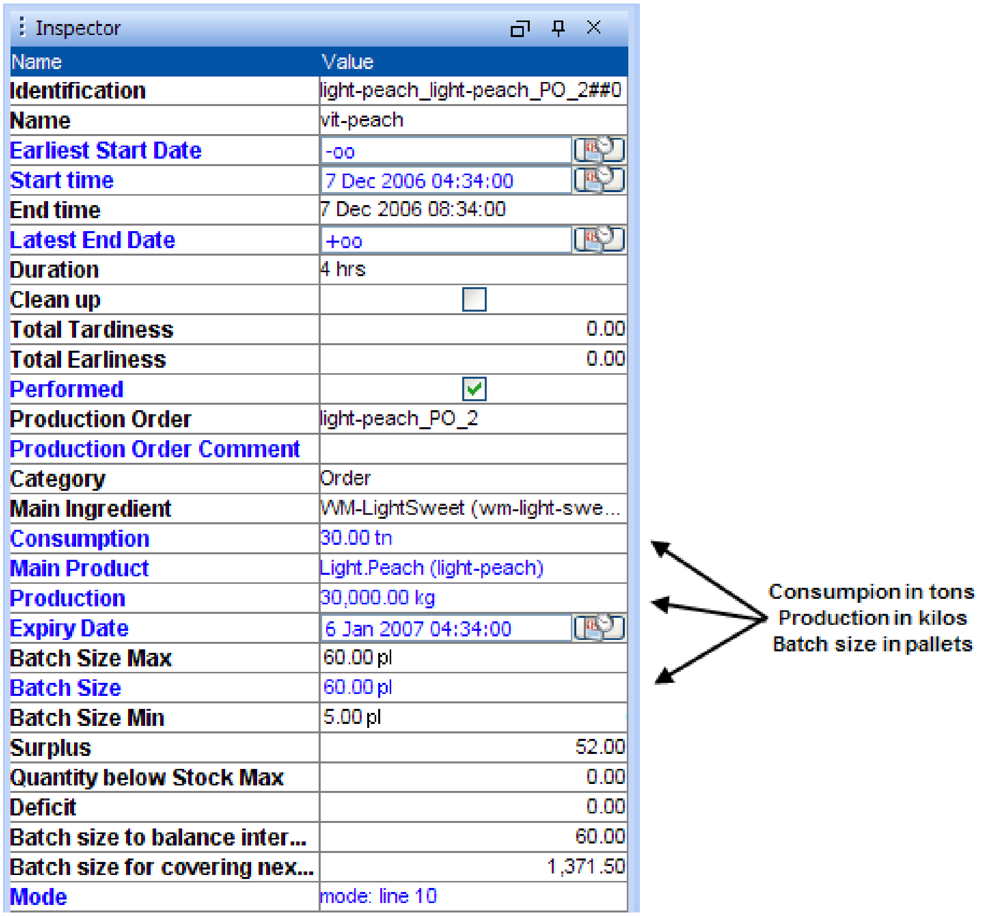
identifier	name	family	initial stock	pegging strategy	maturity
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	
bio-prune	Bio.Prune	Bio Fat	250.00	Static	
bio-kiwi-cereals	Bio.Kiwi Cereals	Bio Fat	250.00	Static	
bio-muesli	Bio.Muesli	Bio Fat	250.00	Static	
bio-soy-red-fruits	Bio.Soy Red Fruits	Bio Soy	250.00	Static	
bio-soy-natural	Bio.Soy Natural	Bio Soy	250.00	Static	
bio-skim-kiwi	Bio.Skim Kiwi	Bio Skim	250.00	Static	
bio-skim-prune	Bio.Skim Prune	Bio Skim	250.00	Static	

Editing data in the GUI

There are numerous enhancements to your ability to edit data in the PPO GUI.

Production order editing with the Inspector

You can adjust the batch size of a production order by changing the quantity produced or the quantity consumed directly in the inspector. Moreover, the batch size, consumption, and production can be specified in different units if enabled in the model (secondary units for material declared). The batch size is updated automatically when fields are edited.



Name	Value
Identification	light-peach_light-peach_PO_2##0
Name	vit-peach
Earliest Start Date	-oo
Start time	7 Dec 2006 04:34:00
End time	7 Dec 2006 08:34:00
Latest End Date	+oo
Duration	4 hrs
Clean up	☐
Total Tardiness	0.00
Total Earliness	0.00
Performed	✓
Production Order	light-peach_PO_2
Production Order Comment	
Category	Order
Main Ingredient	WM-LightSweet (wm-light-swe...
Consumption	30.00 tn
Main Product	Light.Peach (light-peach)
Production	30,000.00 kg
Expiry Date	6 Jan 2007 04:34:00
Batch Size Max	60.00 pl
Batch Size	60.00 pl
Batch Size Min	5.00 pl
Surplus	52.00
Quantity below Stock Max	0.00
Deficit	0.00
Batch size to balance inter...	60.00
Batch size for covering nex...	1,371.50
Mode	mode: line 10

Master data editing in the GUI

The master data available for editing in the GUI continues to expand. Use the **Edit** selection on the contextual menu on the **Master Data** tables to display a dialog box that allows you to see and modify your model data.

You can edit the secondary units of materials on the **Advanced Options** tab of the **Material** editing window.

On the recipe editor window, you can edit the secondary resource requirements of an activity mode.

Setup activities and constraints are now visible on the recipe editing window.

Two new tabs on the **Master Data** view give you access to **Units of Measure** and **Bucket Sequences**. Both contain editable data. You can add or remove bucket templates, with the effects immediately visible on a time scale in the editor window. You can create buckets the same way when generating a new model with the model wizard (**File -> New Model**)

New design and function for some views and tools

Optimization windows

The **Batching advanced options** selection has been removed from the **Optimize the scenario** dialog box. The **Slack on Planned Start Time** and **Slack on Planned End Time** selections have been moved to the **Scheduling advanced options** page.

The **Optimization in Progress** window has been redesigned to provide better access to details of the optimization modules.

Start page

The **Start Page** has a new appearance, and the **Installed Plugins** are listed along with version number.

Stock Coverage view

The **Stock Coverage** view now has different layouts selectable by a combo box, and you can add your own layouts. Each layout corresponds to a different use case: Inventory balance, inventory target, transport layout, demand fulfillment, and available to promise production.

In previous versions of PPO, two modes of inventory consumption by demand were considered; one was called "optimizer result" that used the decision of demand satisfaction of the optimizer as output. The other mode was called FIFO, which always decreased the inventory by the amount of material requested at due date. This last mode was used for interaction but did not satisfy all constraints, such as the availability (the stock could be negative), storage unit compatibility, maturity and shelf life constraints. With PPO 3.2, there is now an automatic repegging to the demand at each interaction that enforces the constraints and may not satisfy the whole demand. The stock displayed can no longer be negative, and no immature or obsolete stock can be consumed by the demand. A red bar indicates the unsatisfied demand. A gray bar indicates the obsolete inventory thrown away at expiration time. An inflow bar indicates the amount of missing intermediate material. This automatic demand repegging is based on a linear optimization taking the scheduling weights into account in the objective function. It pegs the procurements and production orders to the demand while taking the scheduling solution into account. If detailed scheduling is not required by the optimization profile, and for time windows beyond the scheduling horizon, the planned productions from the planning solution are also pegged to the demand.

For more information see *Stock Coverage view*.

Stock Event view

In the **Stock Event** view you can see applicable events that occur before the origin; this includes past procurements that are in fact stock elements or past demand reserving a stock element.

Waste events are automatically created to retract the obsolete material.

Changes with tools

You can now hide the tooltip of activities on the Gantt diagram, using a plugin installer class in Java™ and the method:

```
plantApplication.setShowingTooltipsOnResourceChart(false);
```

A new **Inspector** tool is available for the **Plant Layout** view.

There are new icons and new overall appearance in the PPO 3.2 GUI. The *Online Help* has a full listing of all icons and shortcut keys.

Modeling changes

There are changes in modeling demand used for interactive planning, modeling initial stock, and enforcement of planning mode decisions.

In this section

Promised demand used in interactive planning
"Firm" demand is renamed as *promised demand*.

Modeling initial stock
There is now one unique method of modeling initial stock.

Persistence of planning decisions
Planning decisions are now available in the model, and can be enforced upon scheduling.

Promised demand used in interactive planning

Promised demand (previously known as *firm* demand) is a concept useful for editing a production plan in the GUI. For example, suppose you have optimized a production plan, and are now wondering if it is possible to add more demand and what the effects would be. Use promised demand to indicate demand that is already promised to a customer; such a demand is not included in the "Available-To-Promise" (ATP) calculation in the next optimization. You can then run planning simulations based upon ATP, capable to promise (CTP), and profitable to promise (PTP) calculations. The **Stock Coverage** view has two columns (hidden by default) useful in this context: **ATP** and **Cumulative ATP**.

On the `PPO_DEMAND` table the column `FIRM` has been renamed as `PROMISED`. In the API, the affected methods of `IloMSDemand` have been renamed `setPromised(boolean)` and `isPromised()`.

Note that promised demand is used only in the ATP calculation; also note that marking a demand as promised is not enough to make the demand delivery mandatory for the optimizer. For more information, see *Production planning simulations*.

Modeling initial stock

In PPO V3.2 there is one unique method to enter initial stock into your model, and that is via procurement: Use the `PPO_PROCUREMENT` table or API class `IloMSProcurement`. Any procurement with a reception date before the start of the model is considered to be a stock element. The associated production date is used to compute the age of the stock.

Several model elements are now obsolete, but the CSV reader is still able to read these tables and generate the equivalent elements in the new model. See *Migration to schema version 4.0* for more information.

Obsolete model elements due to this change:

- `PPO_MATERIAL|INITIAL_QUANTITY` and `PPO_STORAGE_UNIT_MATERIAL|INITIAL_QUANTITY`.
- `PPO_STORAGE_TO_xxx_ARC` and `PPO_xxx_TO_STORAGE_ARC` tables.
- `IloMSMaterial.setInitialQuantity(double)` and `IloMSStorageUnit.addStorableMaterial(IloMSMaterial.double)`

Persistence of planning decisions

Decisions taken by the planning engine regarding resource assignment are conserved by production orders, and are available in the `PPO_PRODUCTION_ORDER_PLANNED_MODE` table. You can enforce the scheduling engine to respect the planning decisions made on resource usage with the parameter `modeOfPlanningOnly` in the `PPO_SETTING` table.

Enhancements to planning, batching, and scheduling

Describes new functionality available in the optimization modules for Plant PowerOps (PPO) V3.2.

In this section

Planning module enhancements
Describes improvements in PPO in the areas of dynamic safety stock and infeasibility management. Also see the section *Persistence of planning decisions*.

Batching algorithms
There are three batching algorithms available, and the constraint based algorithm has been greatly improved.

Scheduling
A new scheduling model for pegging arcs is available in PPO V3.2.

Planning module enhancements

Describes improvements in PPO in the areas of dynamic safety stock and infeasibility management. Also see the section *Persistence of planning decisions*.

In this section

Dynamic safety stock
Dynamic fill rate and dynamic service level are innovative approaches in PPO, and are improved for this release.

Infeasibility management
Describes infeasibility management and the multipass planning algorithm.

Dynamic safety stock

The 3.1 release of PPO introduced the ability to define safety stock based on fill rate or cycle service level, depending upon demand and cycle time variability with a standard approach based on average lead time. Also in V3.1, PPO introduced an innovative approach to define safety stock dynamically, depending upon planned lead time (taking into account the planned date of the next production). This allowed users to manage inventory according to safety stock settings in conjunction with manufacturing optimization; previously, one could manage only through direct manipulation of safety stock settings.

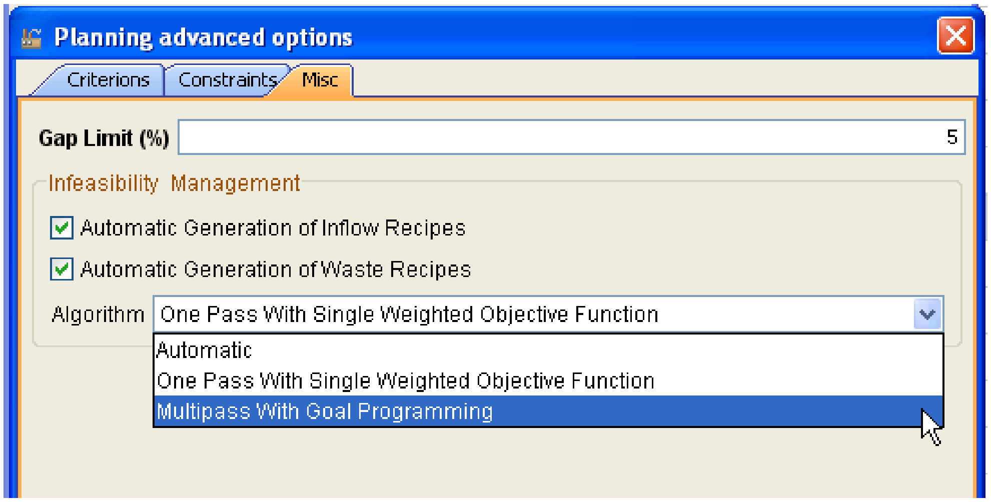
In PPO V3.2, the management of dynamic safety stock has been improved, leading to an average inventory reduction of 25%.

For more information on using this technology, see *Using service levels, lead time, and demand variability to manage stock levels*.

Infeasibility management

With **Infeasibility Management**, you can make selections for automatic generation of inflow or waste recipes, and choose between several planning algorithms.

These selections are available in the relational data model, API, and on the **Planning advanced options** tab of the **Optimize the scenario** dialog box.


Planning advanced options
Criterions
Constraints
Misc

Gap Limit (%) 5

Infeasibility Management

✓ Automatic Generation of Inflow Recipes

✓ Automatic Generation of Waste Recipes

Algorithm One Pass With Single Weighted Objective Function

Automatic
One Pass With Single Weighted Objective Function
Multipass With Goal Programming

The inflow and waste recipes are used to deal with infeasible balances (missing ingredients, perished materials, surplus materials). These artificial recipes introduce slack variables to repair infeasible balance of materials for fixed orders. High costs are generated automatically, so use of these recipes is a fallback option in case the model is infeasible. At each optimization run, the old automatic inflow and waste recipes and orders are discarded, and new ones are generated. The artificial recipes are labeled with a type of “Automatic” (see `PPO_RECIPE|RECIPE_TYPE`).

You can let PPO decide which algorithm to use for planning (**Automatic**) or choose yourself between single pass or multipass.

The single pass approach solves a single mathematical programming model with a single weighted objective function, combining business objectives and "technical" costs. With this approach, the costs of processing these recipes must be carefully computed so that the optimizer uses them only as a fallback position, in case of infeasibility. The drawback of this approach is the potentially wide range in numerical values that coexist during optimization, which can lead to numerical stability issues. The business objectives could get diluted in an objective function containing high technical costs. A bad solution with respect to the business objective may result if the relative gap limit stops optimization when the solution is "good enough."

The multipass approach uses goal programming to deal with infeasible material flow. The business objective is kept separate from technical costs such as inflow and waste recipe costs or nondelivery costs. This approach assumes it is always better to deliver a product, as compared to other considerations.

To summarize, the standard approach with a single pass and a single objective function is not always appropriate. Some considerations of using a single objective function include:

- ♦ Waste and Inflow recipe mode costs are technical costs that must dominate the others;
- ♦ Non delivery costs must dominate the business costs;
- ♦ Business cost contains all the other costs: inventory, inventory deficit, setups, processing costs, and so forth.

Using a single objective function in a single pass may lead to:

- ♦ Numerical problems because of the very different magnitudes of the costs;
- ♦ The gap limit is meaningless because the business costs become negligible as compared to technical costs;
- ♦ The optimizer spends a lot of time before it manages to satisfy the demand.

Using the multipass algorithm will minimize certain goals in the following order:

- ♦ First minimize the business objective with mandatory demand and no inflow/waste;
- ♦ If that fails, then:
  - ● Minimize inflow and waste;
  - ● Minimize nondelivery, enforcing inflow and waste levels as previously found;
  - ● Minimize the business objective, enforcing nondelivery, inflow, and waste levels as found previously.

These choices are also available in the data schema (`PPO_OPTIMIZATION_PROFILE` table) and in the API in the enumeration `IloMSPlanningAlgorithm`.

Batching algorithms

Three batching algorithms are available: Basic heuristic, advanced heuristic, and constraint based. There is also an automatic selection, which means that PPO first tries the heuristic algorithms and if these fail to respect constraints, then the constraint based algorithm is invoked.

The constraint based algorithm has been greatly improved to yield better decision making. Numerical precision issues have been addressed.

Scheduling

The setting `bUseNewSchedulingModelForPeggingArcs` brings greater accuracy to the scheduling of pegging arcs. This may lead to improvements in tardiness and nondelivery in the 1:N (one producer, several consumers) and N:1 cases. In many-to-many relationships involving storage tanks, the new scheduling model allows ending the storage activity at the end of the last consumption than can occur before the end of the latest consuming activity.

For more information see *Parameters for optimization*.

Also, you can enforce planning decisions upon the scheduling engine now; see the section *Persistence of planning decisions* for more information.

Relational model changes

This section introduces the Plant PowerOps (PPO) schema version 4.0. The data schema goes through a major upgrade for PPO V3.2 due to separation of master and transactional data.

In this section

Separation of master data and transactional data
Prototype tables have been added to the data schema.

Migration to schema version 4.0
Describes how to migrate your existing database and csv files to the new schema version.

Data schema modeling changes
Lists the changes to the data schema for PPO V3.2.

Separation of master data and transactional data

The 3.1 release of PPO separated the master data from the transactional data into two corresponding views in the GUI. For V3.2, this process continues in the relational model with the separation of the recipe prototype data from the generated instances of production orders. Many of the recipe prototype tables are identified with a name ending with `_PROTO`, while the corresponding generated instance table names have not changed. For example, some recipe table names are `PPO_RECIPE`, `PPO_ACTIVITY_PROTO`, and `PPO_MODE_PROTO` that model your master data, while the generated production order data resides in tables such as `PPO_PRODUCTION_ORDER`, `PPO_ACTIVITY`, and `PPO_MODE`.

Migration to schema version 4.0

This section provides suggested migration pathways to move your csv and mdb files from a previous version to the PPO schema version 4.0.

Spreadsheet csv files

The suggested process to migrate your csv files to the new schema:

- Open the original csv file in the PPO 3.2 GUI. The data is automatically ported to the new schema.
- Save the scenario as a csv file (named accordingly) to save the migration changes.

Database mdb files

The suggested process to migrate your mdb files to the new schema:

- Open the mdb file with the 3.0 or 3.1 version of the PPO GUI.
- Save the scenario as a csv file type using **File** —> **Save Copy Of Scenario As** and specify ***.csv** in the **Files of type** field.
- Open your saved csv file in the PPO 3.2 GUI. The data is automatically ported to the new schema.
- Save the scenario as a database mdb file to save the migration changes.

Data schema modeling changes

This section lists the changes to the data schema for PPO V3.2. To see the full descriptions for the new fields and tables please refer to the *Plant PowerOps Data Schema*.

Renamed tables

Tables have been renamed in order to better support precedence and compatibility constraints:

- `PPO_ACTIVITY_COMPATIBILITY` has been renamed to `PPO_PROD_PROD_COMPAT`.
- `PPO_PRECEDENCE` has been renamed to `PPO_PROD_PROD_PRECED`.

New tables

Many new tables have been added to support the separation of the prototype information from the generated data, and to support precedence and compatibility constraints:

- `PPO_ACTIVITY_CHAIN_PROTO`
- `PPO_ACTIVITY_PROTO`
- `PPO_ACTIVITY_SETUP_STATE_PROTO`
- `PPO_MATERIAL_PRODUCTION_PROTO`
- `PPO_MODE_PROTO`
- `PPO_PROD_ORDER_PLANNED_MODE`
- `PPO_PROD_PROD_COMPAT_PROTO`
- `PPO_PROD_PROD_PRECED_PROTO`
- `PPO_PROD_SETUP_COMPAT`
- `PPO_PROD_SETUP_COMPAT_PROTO`
- `PPO_PROD_SETUP_PRECED`
- `PPO_PROD_SETUP_PRECED_PROTO`
- `PPO_SECONDARY_RESOURCE_PROTO`
- `PPO_SETUP_ACTIVITY_PROTO`
- `PPO_SETUP_MODE_PROTO`
- `PPO_SETUP_PROD_COMPAT`
- `PPO_SETUP_PROD_COMPAT_PROTO`
- `PPO_SETUP_PROD_PRECED`

- PPO_SETUP_PROD_PRECED_PROTO
- PPO_SETUP_SECONDARY_RES_PROTO
- PPO_SETUP_SETUP_COMPAT
- PPO_SETUP_SETUP_COMPAT_PROTO
- PPO_SETUP_SETUP_PRECED
- PPO_SETUP_SETUP_PRECED_PROTO
- PPO_SPANNING_PROTO

Also the PPO_DATA_SCHEMA table has been added for database persistence to record the schema version of PPO.

Deprecated and removed tables

The following tables have been removed (previously deprecated):

- PPO_SHIPPING_COST_FCT
- PPO_SHIPMENT_TYPE
- PPO_BATCHING_CRITERON_WEIGHT
- PPO_MATERIAL_SHIPMENT_TYPE
- PPO_SHIPMENT
- PPO_SHIPMENT_PRODUCTION_ORDER

The following tables have been deprecated:

- PPO_PROD_TO_STORAGE_ARC
- PPO_STORAGE_TO_PROD_ARC
- PPO_STORAGE_TO_DEMAND_ARC
- PPO_PROCUREMENT_TO_STORAGE_ARC

Changes to columns within tables

This section lists internal changes to tables since PPO 3.1.

Changes to the PPO_MODE table

The following columns of the PPO_MODE table have been renamed:

- FIXED_PROCESSING_TIME_MIN has been renamed to PROCESSING_TIME_MIN
- FIXED_PROCESSING_TIME_MAX has been renamed to PROCESSING_TIME_MAX
- FIXED_COST has been renamed to COST

The following columns have been added to the PPO_MODE table:

- ♦ FIX_PROCESSING_TIME_MIN_PROTO
- ♦ FIX_PROCESSING_TIME_MAX_PROTO
- ♦ VARIABLE_PROCESSING_TIME_PROTO
- ♦ FIXED_COST_PROTO
- ♦ UNPERFORMED_COST_PROTO

The following columns have been removed from the PPO_MODE table:

- ♦ FIXED_PROCESSING_TIME
- ♦ VARIABLE_PROCESSING_TIME
- ♦ VARIABLE_COST

Changes to tables of type: _MATERIAL

From table PPO_MATERIAL, the following columns have been removed:

- ♦ INITIAL_QUANTITY
- ♦ LENGTH
- ♦ WIDTH
- ♦ DEPTH
- ♦ VOLUME
- ♦ WEIGHT
- ♦ INTERNAL_LENGTH_MAX
- ♦ INTERNAL_WIDTH_MAX
- ♦ INTERNAL_DEPTH_MAX
- ♦ INTERNAL_VOLUME_MAX
- ♦ INTERNAL_WEIGHT_MAX

In table PPO_MATERIAL_PRODUCTION:

- ♦ Column VARIABLE_QUANTITY has been removed
- ♦ Column FIXED_QUANTITY has been renamed to PRODUCED_QUANTITY
- ♦ Column VARIABLE_QUANTITY_PROTO has been added
- ♦ Column FIXED_QUANTITY_PROTO has been added

In table PPO_STORAGE_UNIT_MATERIAL:

- ♦ Column INITIAL_QUANTITY has been removed

♦ Column MERGING_LIMIT has been removed

Changes to other table columns

In table PPO_DEMAND the column FIRM has been renamed to PROMISED, and SHELF_LIFE_MIN has been renamed to REMAINING_SHELF_LIFE.

In table PPO_OPTIMIZATION_PROFILE a new column PLANNING_ALGORITHM has been added, and the columns BATCHING_HORIZON and BATCHING_TIME_LIMIT have been removed.

In table PPO_PROD_TO_PROD_ARC the column STORAGE_UNIT_ID has been removed.

In table PPO_PROD_TO_STORAGE_ARC the column STORAGE_UNIT_ID has been removed.

In table PPO_ACTIVITY the column RECIPE_ID has been removed.

In table PPO_STORAGE_TO_DEMAND_ARC the column STORAGE_UNIT_ID has been added.

In table PPO_PRODUCTION_ORDER the column PLANNED_SUB_RECIPE_INDEX has been removed.

In table PPO_STORAGE_TO_PROD_ARC the column STORAGE_UNIT_ID has been added.

In table PPO_PROCUREMENT_TO_DEMAND_ARC the column SHIPMENT_ID has been removed.

Reporting with Tableau

Tableau has been updated to version 4.1 in PPO V3.2. When generating Tableau reports, all dates are now real dates in the time zone of the model, as opposed to integers.

Tableau allows you to generate detailed, customized reports about your problem data. Tableau is integrated as a plug-in to PPO, and is shipped with several default templates. For Tableau documentation either press **F1** when Tableau starts or go to *http://www.tableausoftware.com*. Note that Tableau is enabled for Unicode and is compatible with data stored in any language; however the user interface and supporting documentation are available in English only.

For more information on customizing Tableau within PPO, see *Customizing report generation with Tableau*.

Documentation, acknowledgement

For the 3.2 release, PPO documentation features improved organization and full search in CHM format.

In this section

Publications in the documentation set
Describes the information delivered with Plant PowerOps.

Accessing the CHM, PDF, and HTML documentation
Describes how to access each of the different documentation formats.

Acknowledgement
Use of algorithm for computing the inverse normal cumulative distribution function.

Publications in the documentation set

Information delivered with Plant PowerOps is organized as follows:

♦ The *Release Notes*

  Introduces the new functionality and features of the most recent release.

♦ *Implementation of Plant PowerOps*

  Describes the optimization process, data modeling, optimization criteria and Key Performance Indicators. Serves as a companion reference to the data schema and API manuals.

♦ *Applied use of Plant PowerOps*

  Shows how to use the PPO GUI and data model to handle industry concerns such as managing stock corridors and service levels, and planning using *available-to-promise* calculations. Also includes some information on basic use of the GUI.

♦ *Examples and tutorials*

  Examples of creating data models to represent manufacturing processes, ranging from the basic to complex. Choose from examples using the data schema, C++ API or Java API.

♦ *Reference Documentation*

  Reference information includes the following:

  - *Customizing and extending Plant PowerOps*

    Modifying your PPO installation through customization, configuration changes, and extension by plug-in.

  - *Plant PowerOps Entity Relationship Diagrams*

    Diagrams showing model entities.

  - *Data schema*

    The complete list of data schema tables and fields with concise descriptions.

  - *C++ Reference Manual*

    The C++ API reference for PPO.

  - *Java Reference Manual*

    The Java™ API reference for PPO.

There is additional information available within the PPO GUI on tooltips, messages, and descriptive help text in various windows. The GUI also has the *Online Help* which provides basic information about system requirements, the menu and toolbars of the application, and shortcut keys.

Accessing the CHM, PDF, and HTML documentation

CHM and PDF

CHM is available from the PPO GUI by selecting **Ctrl + D**. You can also launch the documentation from the PPO menu bar: **Help > Documentation.**

You can also access the documentation without starting PPO. For example on Windows® XP select: **Start > All Programs > ILOG > IBM ILOG Plant PowerOps > Documentation Plant PowerOps**.

The CHM and PDF documentation is located within your PPO product installation directory (represented as <*PPOInstallDirectory*>). You can find the CHM files in the folder:

<*PPOInstallDirectory*>`\doc\chm`

The PDF documentation is in the folder:

<*PPOInstallDirectory*>`\doc\pdf`

The naming convention for these files is as follows:

- ♦ **ppodoc**: The core information of the documentation set. In CHM format, this document includes access to all content.
- ♦ **ppocplus**: The *Plant PowerOps C++ Reference Manual*.
- ♦ **ppojava**: The *Plant PowerOps Java Reference Manual*.
- ♦ **pposchema**: The *Plant PowerOps Data Schema*.

If a title does not exist in the folder, then that publication is not available in that format. For example, the *Plant PowerOps Java Reference Manual* is not available as a PDF file.

HTML documentation

The HTML documentation is available at <*PPOInstallDirectory*>`\doc\html\en-US\documentation.html`. The HTML documentation is cross-platform compatible and includes all available publications.

Features and search

The CHM format provides a quick and easy-to-use search tool that functions across the entire PPO documentation set. It is a very thorough search mechanism; you may prefer to put your search term in quotes to enforce a strict search.

PDF files are useful for annotation, search, and printing purposes. As with CHM, each search hit is highlighted. You can download the Adobe Reader product for free from:

http://www.adobe.com/products

Cross-platform HTML contains all PPO documentation. This format is used with standalone browsers on Windows® and other platforms. There is no built-in search function available on the HTML documentation.

Note that search results may vary slightly across the documentation formats. For example, if searching for only part of a file or path name, you may get more useful results by searching the PDF documentation.

Acknowledgement

IBM® ILOG® Plant PowerOps includes code based on Peter J. Acklam's algorithm for computing the inverse normal cumulative distribution function: http://home.online.no/~pjacklam/notes/invnorm.

Implementation of Plant PowerOps

This section describes what you need to know to create a successful model of your plant data in Plant PowerOps (PPO).

In this section

Introduction to planning and scheduling with PPO
This section describes how you can use PPO to meet the challenges of production planning (PP) and detailed scheduling (DS) in your manufacturing environment.

Optimization and KPIs
This section describes the optimization process that finds a solution to your manufacturing decision problem. There are three optimization modules: planning, batching, and scheduling. You can apply Key Performance Indicators to guide optimization.

The basic data model
This section describes the fundamental elements of a data model for Plant PowerOps. This information helps you populate a problem model in the context of a given application.

Extended use of the data model
This section serves as a continuation of *The basic data model*, describing additional modeling considerations and possibilities of Plant PowerOps. The modeling extensions presented here can improve the accuracy and precision of your model. However, some of these features may make the problems more complex to solve, so it is recommended to use only those that significantly improve the plan.

Introduction to planning and scheduling with PPO

This section describes how you can use PPO to meet the challenges of production planning (PP) and detailed scheduling (DS) in your manufacturing environment.

In this section

Overview
An overview of the challenges in the industry.

What Is IBM ILOG Plant PowerOps?
IBM® ILOG® Plant PowerOps (PPO) is a powerful tool for creating manufacturing rough plans and detailed schedules.

How Plant PowerOps solves the problem
Plant PowerOps integrates planning and scheduling.

Overview

You know the challenges: Unpredictable availability of raw material and unpredictable energy costs. Widely dispersed suppliers, production facilities, and storage facilities. Demand for green technology. Greater product differentiation than ever before (more SKUs), simultaneously with greater volatility in customer demand. All this, in the midst of striving for greater efficiency to compete in the global marketplace.

Among all industry sectors, those that utilize *batch processing* are some of the most capital intensive and capacity constrained. Batch processing includes the food and beverage, consumer packaged goods, pharmaceutical, and chemical manufacturing industries, where volumes are typically high and plant efficiency must be at a maximum level. These industries have already reached high levels of efficiency, but during latter years plant managers have been asked to achieve even greater efficiencies. In these industries a small percentage increase in plant efficiency can bring a large reduction in operating costs. Often, the most cost-effective and potent method of improving plant productivity is to optimally plan and schedule the production resources (including production, maintenance, and quality control resources). However, given the unique characteristics of batch processing industries, human planners and schedulers cannot reach higher levels of efficiency using traditional planning and scheduling systems. That's where IBM® ILOG® Plant PowerOps comes in: To provide plant planners and schedulers the optimized production plans and schedules that take company operations to previously unreachable levels of efficiency.

This publication describes how to implement IBM ILOG Plant PowerOps (PPO) at your facility. It introduces how to model data for PPO, the PPO optimization process, and how to use PPO to solve your manufacturing planning and scheduling problems.

What Is IBM ILOG Plant PowerOps?

PPO is a powerful tool for creating manufacturing rough plans and detailed schedules. You'll find PPO vital for creating both long term production plans and detailed, optimized manufacturing schedules. PPO has fully integrated planning and scheduling. You define the problem once in the PPO model, and PPO plans the volumes, production orders, and schedules using the planning, batching, and scheduling modules.

PPO is based on the standard production model used in batch processing industries: the *Recipe Model*. The recipe model strictly integrates material data and process data (contrasted to discrete manufacturing industries which use the *data model* standard which decouples materials data from process data). PPO is a third generation APS product, based on the most advanced optimization engines available on the market. Prior to suggesting a plan or a schedule to the user, PPO explores thousands or millions of possibilities in a very short period of time.

Planning and scheduling in batch processing industries is a challenging task. First, a feasible schedule must satisfy a variety of constraints including:

- ♦ Temporal constraints
- ♦ Resource assignment and capacity constraints
- ♦ Shelf life and maturity constraints
- ♦ Limited capacity tanks/silos
- ♦ Quality control constraints
- ♦ Equipment connection constraints
- ♦ High material flow synchronization between upstream and downstream equipment
- ♦ Variable standard efficiencies
- ♦ Energy constraints
- ♦ Changeovers and Cleaning In Place (CIP) constraints

In addition, an appropriate schedule must respond to a number of potentially conflicting optimization criteria. It must:

- ♦ Respond to all or at least to the most important customer demands and production orders driven by those demands;
- ♦ Deliver the products on time (that is, not so late that they cannot be delivered to the customer on time, and not so early that the goods require storage);
- ♦ Manufacture the products at the lowest possible cost while managing inventory to optimized levels;
- ♦ Account for variability in demand and lead time in order to achieve a targeted service level;
- ♦ Avoid lengthy and costly machine setups while allowing time slots for cleaning in many industries.

These constraints and optimization criteria, and more, can all be expressed using PPO.

How Plant PowerOps solves the problem

PPO optimization components

The optimization algorithms provided in PPO are based on proprietary state-of-the-art optimization technology. They automatically perform the difficult task of establishing the most appropriate trade-offs between various cost factors to generate feasible, low-cost production schedules.

Optimization occurs within three modules in PPO: Planning, Batching, and Scheduling. The PPO optimization planning engine has a mixed integer and linear approach based on IBM® ILOG® CPLEX® . This approach captures the trade-off between the supply chain objective (keeping the right amount inventory to avoid stock deficit or waste) and the manufacturing objectives (such as minimizing setups and cleanups). The PPO optimization scheduling engine features a constraint-based scheduling approach *dedicated* to manufacturing scheduling and is based on IBM ILOG CP. These modules are described in greater detail throughout the documentation.

PPO visualization components

In PPO, the optimization algorithms are integrated with an interactive graphical user interface (GUI). The PPO GUI provides many views of the scheduling problem as well as the plans, inventory, and resource utilization data generated by the PPO optimization engine. The graphical planning board is built with IBM ILOG JViews Gantt Chart. These views can be displayed simultaneously and are automatically updated at each user interaction. The impact on demand coverage of resizing or delaying an order in the Gantt is seen immediately.

The **Gantt Diagram** displays the best solution found by PPO, and you can edit the problem data. You can change constraints, resources, criteria and Key Performance Indicators, and regenerate a new plan. With several plans displayed in the GUI, you can easily compare the results.

Each plan has multiple views to enable users to view data in tables, graphs, and in Gantt chart format. Importantly, PPO is interactive, allowing users to manually override the generated plan. For example, you can:

- Modify orders data
- Change production sequences
- Simulate adding overtime or extra shifts, as well downtime periods
- Modify the efficiency or capacity of resources.
- Postpone or anticipate activities of a production order
- Change resource assignments
- Fix or firm production orders
- Manage multiple scenarios.

There is immediate graphical rescheduling to help the user interactively change the scheduled activities, while maintaining a certain level of consistency.

Optimization and KPIs

This section describes the optimization process that finds a solution to your manufacturing decision problem. There are three optimization modules: planning, batching, and scheduling. You can apply Key Performance Indicators to guide optimization.

In this section

Optimization modules and solution
Describes the production planning, batching, and detailed scheduling modules of PPO.

Key Performance Indicators
Key performance indicators are used to affect the optimization process and to track various costs and revenue.

Optimization profile
Optimization profiles are used to contain parameters that tune or direct the optimization engines.

Advanced optimization
Describes various considerations regarding optimization, including advanced options, post processing, parameters, complexity considerations and problem decomposition.

Optimization modules and solution

Planning, batching, and scheduling present different programming challenges, and so require that the PPO optimization modules use different technologies to find an optimal solution. By default, the three modules are called in sequence, passing results from one to the next. If desired, you can create a more complex decomposition by using a plug-in (see *Customizing and Extending PPO*).

Generally, after selecting the **Optimize** button in the PPO GUI or calling the `solve` method on the `IloMSModel` class:

- ♦ The production planning module infers an abstraction of the scheduling problem transparently, and determines (under finite capacity constraints) the volume to produce per time bucket while optimizing criteria.
- ♦ Then the batching engine creates production orders respecting batch size constraints and the material flow.
- ♦ The scheduling engine assigns to the production order activities the *modes*, *start times*, and *end times* that satisfy the constraints. Along with the detailed schedule you can find data regarding resource utilization, inventory levels, bottlenecks, and much more.
- ♦ Finally, the demand repegging algorithm is called to create the final arcs to demand. If *Post processing* options have been selected, then those modules are called.

The production planning module

The PPO production planning module generates rough production plans as well as lot recommendations if appropriate, while taking into account operational level constraints such as multidimensional changeovers, capacity limitations, inventory costs, push and pull production strategies, and so forth. It also provides replanning capabilities, allowing the planner to manually change the plan, partially freeze portions of the plan, and automatically replan.

Based on the data model, this module divides the time line into appropriate time “buckets” and creates the production plan by determining for each bucket the amount of each recipe to execute, and the amount of each demand to satisfy. Note that it may be desirable for you to define the length of time buckets yourself, see *Defining time buckets* for more information.

The incentive to create material production plans comes from *nondelivery costs* and *revenue*. Nondelivery costs on demands are mandatory in order to provide an incentive for optimization to fulfill those demands. An infinite nondelivery cost is equivalent to a mandatory demand. Revenue represents the gain achieved by satisfying customer demand, and provides additional incentive to produce final products. Without either nondelivery costs or revenue, the planning module may decide not to produce anything (unless target stock has been defined). See *Costs and revenue* for more on that subject. In the PPO GUI, you may notice an additional component of the global objective called *perturbation*. This is a technical cost that the planning engine uses to break the symmetries of the problem

The planning results are called *planned productions*. This data is displayed in the Plant PowerOps GUI, which enables the planner to directly edit the plan. Planning results are available in several views, most concisely on the **Planned Productions** tab in the **Transactional Data** view, shown below. For those working with data tables, the planning results are stored in several tables: `PPO_PLANNED_PRODUCTION`, `PPO_PLANNED_DELIVERY`, and

PPO_PLANNED_PRODUCTION_MODE. Decisions of resource assignment taken by the planning engine are conserved by the production orders, and are available in the table PPO_PRODUCTION_ORDER_PLANNED_MODE.

The batching module

The PPO batching module generates production orders and, more generally, structures the overall material flow from raw material procurement to the satisfaction of demands for final products. Batching is the module that links planning to scheduling.

Given the results of the production planning, the batching module computes a list of production orders, and pegs them together as necessary (for example, when an order produces intermediate products consumed by other orders). The batching module also pegs the production orders with the specific demands. The batching results are available in the GUI relational tables, most concisely on the **Production Orders** and **Arcs** tables in the **Transactional Data** view, as shown below. For those working with data tables, the production orders are stored in the PPO_PRODUCTION_ORDER table, and the pegging relations (arcs) are stored in arc tables such as PPO_PROD_TO_PROD_ARC and PPO_PROD_TO_DEMAND_ARC.

The data model and GUI allow for the choice between several different engines to perform batching. The *Heuristic* engine batches first, then pegs the production orders. *Advanced Heuristic* is "smarter" than the simple heuristic in the sense it batches on the fly while pegging orders. *Constraint Based* uses constraint programming to solve batching, and *Automatic* tries successively the previous engines, until one succeeds. The constraint based batching engine provides improvements in batching and pegging production orders, demands, and procurements when complex pegging cardinalities are involved.

The scheduling module

The PPO scheduling module allocates production orders to resources, and builds detailed schedules with precise start and end times. It handles complex manufacturing constraints such as resource compatibility constraints, resource calendars, multidimensional changeovers and cleaning, while maximizing resource efficiency and service levels. It also provides rescheduling capabilities, allowing the planner to manually change the schedule, partially freeze portions of the schedule, and automatically reschedule

For each production order and each activity of the corresponding recipe, the scheduling module selects a production mode, a start time and an end time. The Plant PowerOps GUI allows the planner to visualize and edit the schedule in the **Gantt Diagram**. The scheduling results also appear in the **Transactional Data** view, **Scheduled Activities** tab. The evolution of inventories over time can be viewed in the **Stock Event** and **Stock Coverage** views. The data table results are stored in the PPO_SCHEDULED_ACTIVITY table.

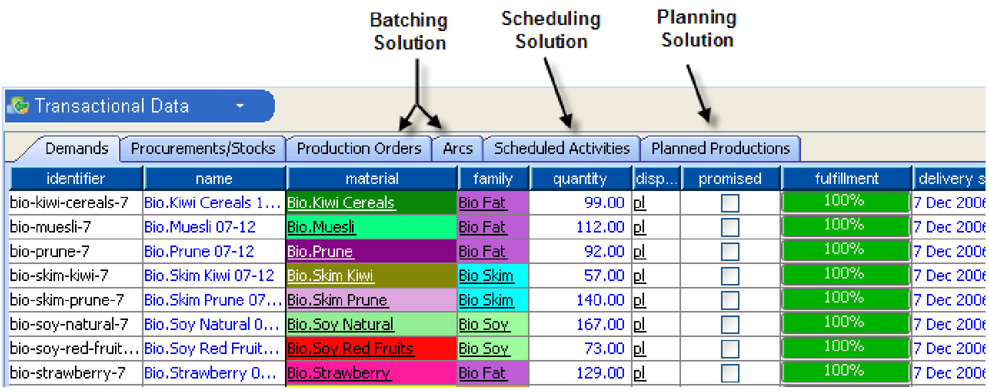
Note that the scheduling engine does not necessarily use the exact resource allocation decided upon by the planning module, as scheduling observes constraints at a finer gradation (time unit as opposed to time bucket). You can, however, enforce the planning decisions with respect to resource allocation upon the scheduling engine (mode of planning); see *PPO parameters*.

Demand repegging

Demand repegging is an optimization-based algorithm that pegs production orders and procurements to independent demands. The result of this module is the creation of arcs to demand. If scheduling is not required by the optimization profile, or during time windows when the planning horizon extends past the scheduling horizon, repegging pegs the planned productions to the demand. Note that demand repegging is called at each interaction even if not required by the optimization profile. In the event of a production shortage, demand repegging will leave as unsatisfied those demands which have the smallest impact on the total cost. Scheduling weights, such as total nondelivery cost, are taken into account during optimization.

Optimization solution in the PPO GUI

The following image shows optimization results in the **Transactional Data** view. More solution data can be found on the **Gantt Diagram** and other views mentioned previously. For example, resource activity, availability, batch sizes and production time per bucket are available in the **Planning Sheet** or **Workload Table** views, and the **Stock Summary** view provides details on materials.



identifier	name	material	family	quantity	disp...	promised	fulfillment	delivery s
bio-kiwi-cereals-7	Bio.Kiwi Cereals 1...	Bio.Kiwi Cereals	Bio Fat	99.00	pl	☐	100%	7 Dec 200
bio-muesli-7	Bio.Muesli 07-12	Bio.Muesli	Bio Fat	112.00	pl	☐	100%	7 Dec 200
bio-prune-7	Bio.Prune 07-12	Bio.Prune	Bio Fat	92.00	pl	☐	100%	7 Dec 200
bio-skim-kiwi-7	Bio.Skim Kiwi 07-12	Bio.Skim Kiwi	Bio Skim	57.00	pl	☐	100%	7 Dec 200
bio-skim-prune-7	Bio.Skim Prune 07...	Bio.Skim Prune	Bio Skim	140.00	pl	☐	100%	7 Dec 200
bio-soy-natural-7	Bio.Soy Natural 0...	Bio.Soy Natural	Bio Soy	167.00	pl	☐	100%	7 Dec 200
bio-soy-red-fruit...	Bio.Soy Red Fruit...	Bio.Soy Red Fruits	Bio Soy	73.00	pl	☐	100%	7 Dec 200
bio-strawberry-7	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	129.00	pl	☐	100%	7 Dec 200

Tuning optimization processes

There are numerous controls you have over optimization. You can set module time limits and optimization criteria; choose time bucket size and algorithms for planning and batching; and apply a variety of parameters and constraints.

Refer to the following topics for more information.

- *Optimization criteria* and *Optimization profile*.
- In the Data Schema, tables such as `PPO_OPTIMIZATION_PROFILE`, `PPO_MODEL`, `PPO_BUCKET`, `PPO_PLANNING_CRITERION_WEIGHT`, and `PPO_SETTING`.
- In the API documentation, classes such as `IloMSOptimizationProfile`, `IloMSModel`, `IloMSBucket`, `IloMSBatchingEngine`, `IloMSPlanningEngine`, and `IloMSSchedulingEngine`.
- *Advanced optimization* topics, especially *Production planning advanced options* and *Detailed scheduling advanced options*.

Key Performance Indicators

Key performance indicators are used in PPO to track particular costs and revenue in the optimized solution; many KPIs can also be used to affect or guide the optimization process. There are several classes of KPIs in PPO:

- Standard KPIs are predefined in the product and are used for tracking purposes. The user can select which of these KPIs to visualize.
- *Optimization criteria* (weighted criteria) constitute a subclass of the standard KPIs. Optimization criteria are commensurable indicators that can be directly taken into account and balanced one against the other by the Plant PowerOps optimization algorithms. These indicators can be prioritized by assigning different weights to each, and they can be conveniently grouped together to form specific *Optimization profile*. This enables the user to easily obtain different solutions depending on, for example, a desire to minimize inventory costs or late delivery costs.
- *Custom KPIs* are not predefined in the product but can be added to meet the demands of any particular plant, process, or industry. Custom KPIs are not taken into account in the objective function of the optimizer. They are computed on a solution to judge its quality with respect to specific business objectives.

Optimization criteria

There are a number of optimization criteria available for your use. Note that not all criteria are available to (or make sense to use with) both the planning and scheduling modules; for example, the setup cost is a pure scheduling cost, and resource and idle costs are used by the planning engine to balance resource usage. Also, when only a planning solution is optimized, the value of some of these criteria must be approximated (since the exact value depends upon the detailed schedule created by the scheduling engine).

The available optimization criteria are described here in terms of the PPO data schema. However, the same concepts apply whether using a database, one of the APIs, or selecting the criterion from the **Optimize the scenario** dialog box in the PPO GUI.

- Total Nondelivery Cost: For each demand, the `PPO_DEMAND` table provides a `NON_DELIVERY_VARIABLE_COST` per unit of material that is not delivered to the customer. The total nondelivery cost is the sum, over all demands, of this `NON_DELIVERY_VARIABLE_COST` multiplied by the number of units of the demand that are not delivered to the customer. *It is mandatory to assign a nondelivery cost in order to provide an incentive for the optimizer to fulfill the demand.* An infinite nondelivery cost is equivalent to a mandatory demand. Note that when a demand is not satisfied, there are two potential costs: A nondelivery cost and a lack of revenue.
- Total Revenue: For each demand, the `PPO_DEMAND` table provides `REVENUE` per unit of material delivered to the customer. The total revenue is the sum, over all demands, of this `REVENUE` multiplied by the number of units of material shipped from inventory to fulfill a demand.
- Total Processing Cost: For each mode, the `PPO_MODE_PROTO` and `PPO_MODE` tables provide a `FIXED_COST` and a `VARIABLE_COST`. For a given production order with batch size `B`, the cost of executing the corresponding activity in this mode is computed as `FIXED_COST` + `B` * `VARIABLE_COST`. The total processing cost is the sum, over all activities, of the cost of the selected modes.

- Total Unperformed Cost: For each mode, the PPO_MODE_PROTO and PPO_MODE table provide an UNPERFORMED_COST of leaving an activity "unperformed" but planned to be executed in this mode. The total unperformed cost is the sum, over all activities, of the cost of leaving activities unperformed.
- Total Setup Cost: For each setup (transition between states of the same setup feature), the PPO_SETUP_MATRIX table provides a SETUP_COST. The total setup cost is the sum, over all transitions and over all features of a given plan, of this SETUP_COST. The setup cost is a scheduling cost and has no effect in planning.
- Total Cleanup Cost: The PPO_MODE_PROTO and PPO_MODE tables provide the processing costs of recipes; for cleanup recipes the processing cost is the cleanup cost. The total cleanup cost is the sum of the processing cost for all cleanups of all resources.
- Total Earliness Cost: For each demand due date, the PPO_DUE_DATE table provides a EARLINESS_FIXED_COST and a EARLINESS_VARIABLE_COST. For each unit of the demand that is delivered E units of time early, the earliness cost of this unit is computed as EARLINESS_FIXED_COST + E * EARLINESS_VARIABLE_COST. The total earliness cost is the sum, over all units of all demands that are delivered early, of this earliness cost.
- Total Tardiness Cost: For each demand due date, the PPO_DUE_DATE table provides a TARDINESS_FIXED_COST and a TARDINESS_VARIABLE_COST. For each unit of the demand that is delivered T units of time late, the tardiness cost of this unit is computed as TARDINESS_FIXED_COST + T * TARDINESS_VARIABLE_COST. The total tardiness cost is the sum, over all units of all demands that are delivered late, of this tardiness cost.
- Total Waste Cost: The waste criterion is used in the material rebalancing engine to penalize the excess of semi-finished materials to be thrown away due to a mismatch between semi-finished and finished products. In planning, the waste criterion is used to penalize the amount of material discarded due to expired shelf life, enforced by *waste recipes*. The total waste cost is defined for planning as the sum of the processing costs of all "planned productions" of waste recipes.
- Total Resource Cost: For each resource and each point in time, the PPO_RESOURCE_CAPACITY_COST table references a piece-wise linear function, allowing computation of the cost of using resource capacity. The purpose of this criterion is to allow you to balance resource usage in planning; it should not be used in scheduling. The total resource cost is the sum, over all resources and over time, of the obtained cost.
- Total Idle Cost: For each resource and each point in time, the PPO_RESOURCE_CAPACITY_COST table references a piece-wise linear function, allowing computation of the cost of leaving resource capacity unused. The purpose of this criterion is to allow you to balance resource usage in planning; it should not be used in scheduling. The total idle cost is the sum, over all resources and over time, of the obtained cost.
- Total Inventory Cost: For each material and each point in time, the PPO_INVENTORY_MAX_COST table references a piece-wise linear function, allowing computation of the cost of maintaining a given quantity of the material in inventory. The total inventory cost is the sum, over all materials and over time, of the obtained cost. Note that PPO provides other mechanisms for managing inventory that may be more appropriate for your facility: Setting a days of supply target, or through use of service levels and production lead time.
- Total Inventory Deficit Cost: For each material and each point in time, the PPO_INVENTORY_MIN_COST table references a piece-wise linear function, allowing computation of a penalty for not maintaining a given quantity of the material in inventory.

The total inventory deficit cost is the sum, over all materials and over time, of the obtained cost. Note that PPO provides other mechanisms for managing inventory that may be more appropriate for your facility: Setting a days of supply target, or through use of service levels and production lead time.

Custom KPIs

Examples of custom KPIs can include:

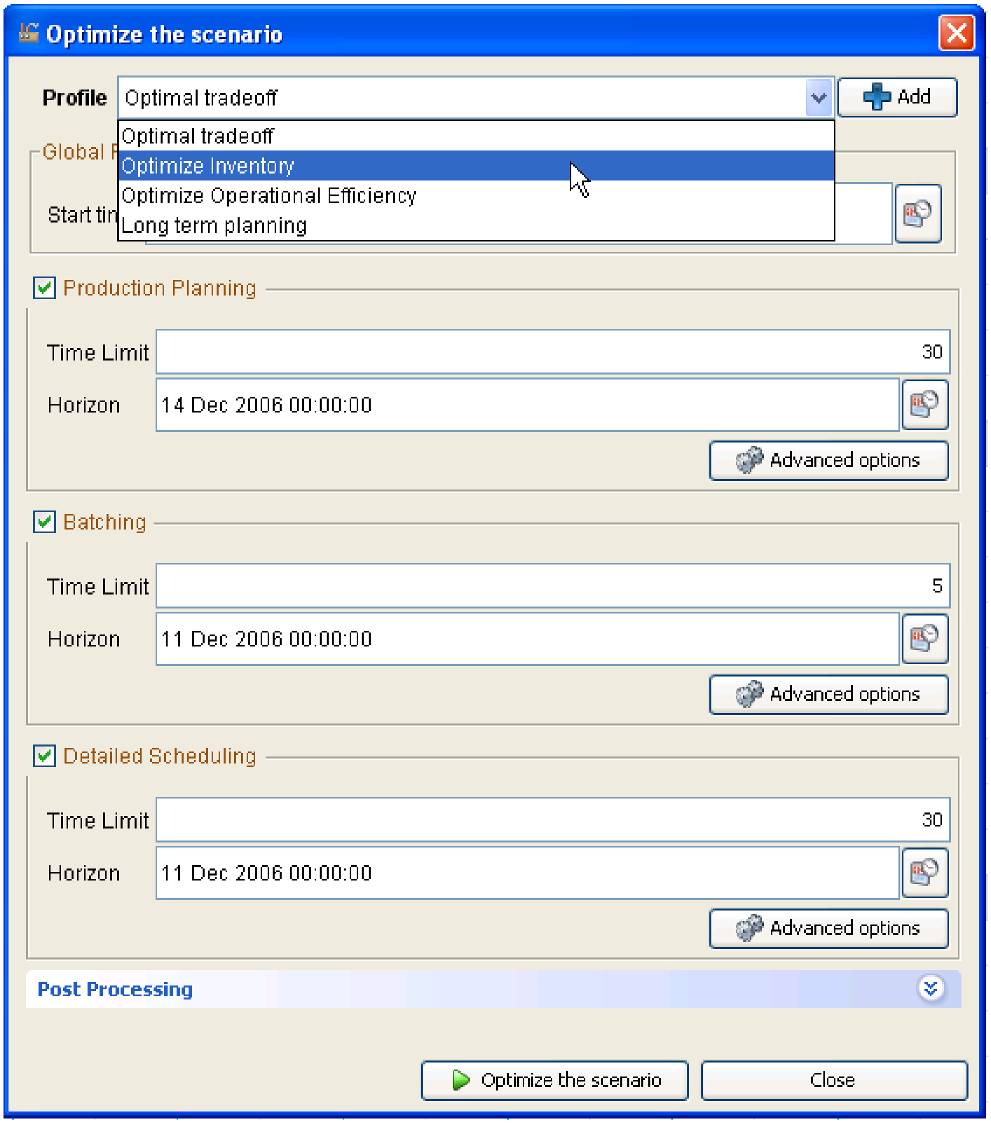
- Average cleaning frequency for a given set of resources.
- Throughput at a given step of the process.
- Average inventory at a given step of the process.
- Number of tardy deliveries of a given class of customers.
- Resource operational efficiency: *Operational time* divided by the *net production time*.
- Resource operational utilization: *Operational time* divided by the calendar time.

Operational time of a given set of resources is the calendar time without bank holidays, idle time, or research and development trials on the resource.

Net production time of a given resource is the output of the production divided by the maximal speed of the resource (expressed as a quantity per unit of time).

Optimization profile

Optimization profiles are used to contain parameters that tune or direct the optimization engines. An optimization profile can ensure that you have a steady set of criteria ready to solve your model data in a consistent way over time. For example, you could create pure planning profiles, or emphasize manufacturing objectives such as minimizing setups and cleanups, or focus on supply chain objectives to respect inventory targets. You could create profiles to optimize for these objectives, and access them in the GUI as shown in this image:



You can control all of the following with an optimization profile:

- ♦ Which modules are included in the optimization;
- ♦ The horizon (end time limit) for the production and scheduling modules;
- ♦ The time limit allowed to the production and scheduling engines for solving;
- ♦ Optimization criteria (such as `TotalSetupCost` as described in *Optimization criteria*) and their associated weights (values) used by planning and scheduling;
- ♦ Other advanced options, such as constraints and gap limit in planning, or which batching algorithm is used;
- ♦ Use of the post processing options such as *material rebalancing*.

In the data schema, the PPO_OPTIMIZATION_PROFILE and PPO_CRITERION_WEIGHT tables are used to control aspects of optimization profiles. In the API, start with the classes `IloMSOptimizationProfile` and `IloMSOptimizationCriterion`.

Advanced optimization

Describes various considerations regarding optimization, including advanced options, post processing, parameters, complexity considerations and problem decomposition.

In this section

Production planning advanced options
Describes advanced options for the planning module.

Detailed scheduling advanced options
Describes advanced options for the scheduling module.

Constraints
Describes constraints enforced by the three modules.

Post processing
Describes post processing options such as material rebalancing.

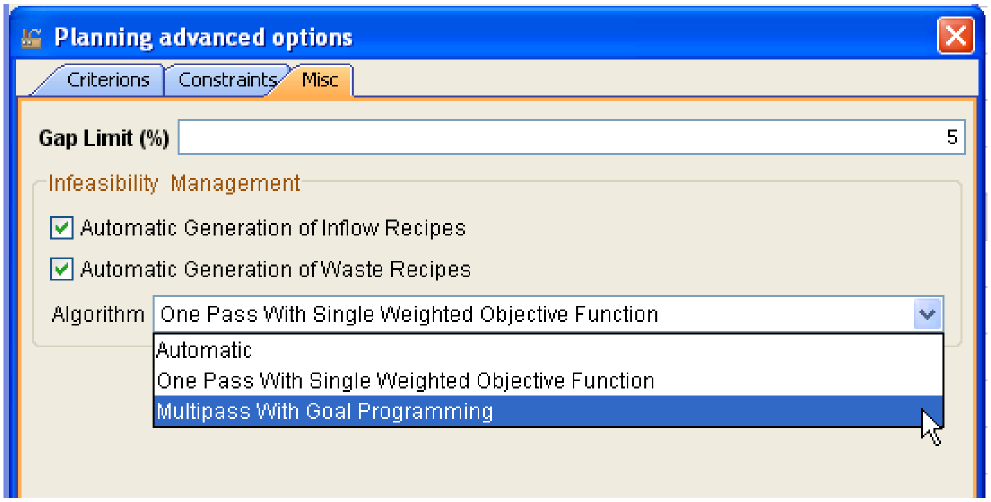
Parameters for optimization
Optimization parameters can be used to affect the solution process.

Complexity considerations
Factors that introduce complexity to the planning module.

Problem decomposition
Methods available to simplify the problem model.

Production planning advanced options

Planning advanced options on the **Optimize the scenario** dialog box include **Criterions**, **Constraints**, and on the **Misc** tab, the ability to alter settings for the **Gap Limit** and **Infeasibility Management**. Criterions are described in *Optimization criteria*. The *Constraints* tab lists hard constraints for the planning module.


Planning advanced options
Criterions
Constraints
Misc

Gap Limit (%) 5

Infeasibility Management

✓ Automatic Generation of Inflow Recipes

✓ Automatic Generation of Waste Recipes

Algorithm One Pass With Single Weighted Objective Function

Automatic
One Pass With Single Weighted Objective Function
Multipass With Goal Programming

Under the miscellaneous settings, the **Gap Limit** sets a stop used on planning optimization. For example, if the gap limit is 1%, then planning will stop as soon as it is proven that the cost of the current solution is *at most* 1% above the optimum. See *CPLEX parameters* for more information.

With **Infeasibility Management**, you can make selections for automatic generation of inflow or waste recipes, and choose between several planning algorithms.

The inflow and waste recipes are used to deal with infeasible balances (missing ingredients, perished materials, surplus materials). These artificial recipes introduce slack variables to repair infeasible balance of materials for fixed orders. High costs are generated automatically, so using these recipes is a fallback in case the model is infeasible. At each run, the old automatic inflow and waste recipes and orders are discarded, and new ones are generated. The artificial recipes are labeled with a type of “Automatic” (see `PPO_RECIPE|RECIPE_TYPE`).

You can let PPO decide which algorithm to use for planning (**Automatic**) or choose yourself between single pass or multipass.

The single pass approach solves a single mathematical programming model with a single weighted objective function, combining business objectives and "technical" costs. With this approach, the costs of processing these recipes must be carefully computed so that the optimizer uses them only as a fallback position, in case of infeasibility. The drawback of this approach is the potentially wide range in numerical values that coexist during optimization, which can lead to numerical stability issues. The business objectives could get diluted in an objective function containing high technical costs. A bad solution with respect to the business objective may result if the relative gap limit stops optimization when the solution is "good enough."

The multipass approach uses goal programming to deal with infeasible material flow. The business objective is kept separate from technical costs such as inflow and waste recipe costs or nondelivery costs. This approach assumes it is always better to deliver a product, as compared to other considerations.

To summarize, the standard approach with a single pass and a single objective function is not always appropriate. Some considerations of using a single objective function include:

- Waste and Inflow recipe mode costs are technical costs that must dominate the others;
- Non delivery costs must dominate the business costs;
- Business cost contains all the other costs: inventory, inventory deficit, setups, processing costs, and so forth;

Using a single objective function in a single pass may lead to:

- Numerical problems because of the very different magnitudes of the costs;
- The gap limit is meaningless because the business costs become negligible as compared to technical costs;
- The optimizer spends a lot of time before it manages to satisfy the demand.

Using the multipass algorithm will minimize certain goals in the following order:

- First minimize the business objective with mandatory demand and no inflow/waste;
- If that fails, then:
    - Minimize inflow and waste;
    - Minimize nondelivery, enforcing inflow and waste levels as previously found;
    - Minimize the business objective, enforcing nondelivery, inflow, and waste levels as found previously.

These choices are also available in the data schema (`PPO_OPTIMIZATION_PROFILE` table) and in the API in the enumeration `IloMSPlanningAlgorithm`.

Note that you can enforce that the scheduling engine must respect planning decisions with respect to resource assignment; see *Parameters for optimization*.

Detailed scheduling advanced options

Scheduling advanced options on the **Optimize the scenario** dialog box include **Criterions**, **Constraints**, and on the **Misc** tab, the ability to set slack on planned start and end times. Criterions are described in *Optimization criteria*. The *Constraints* tab lists hard constraints for the scheduling module.

The choices on the **Misc** tab allow you to assign some flexible time for the scheduling module with regards to decisions made by the planning engine. **Slack on Planned Start Time** is an integer value that defines the number of time units of *anticipation* allowed to the scheduling engine with respect to the start of the time bucket chosen by the planning engine. **Slack on Planned End Time** is an integer value that defines the number of time units of *delay* allowed to the scheduling engine with respect to the end of the time bucket chosen by the planning engine.

Two more advanced scheduling options that are not available on the **Optimize the scenario** dialog box are the settings `modeOfPlanningOnly` and `bUseNewSchedulingModelForPeggingArcs`. For more information see *Parameters for optimization*.

These options are also available as parameters in the data schema (`PPO_SETTING` table) and API (classes `IloMSModel` of `IloMSOptimizationProfile`). Note that you can enforce that the scheduling engine must respect planning decisions with respect to resource assignment;

Constraints

Different hard constraints are taken into account by all three optimization modules. The following list of constraints includes the module which enforces each (**P**lanning, **B**atching, or **S**cheduling).

- Resource capacity (at time bucket granularity for planning, time unit granularity for scheduling)
- Minimal and maximal batch size constraints (P and B)
- Batch size integrality constraints (P and B)
- Setup time constraints (approximated in planning, detailed in scheduling)
- Cleanup constraints (approximated in planning, detailed in scheduling)
- Calendar constraints (P and S)
- Resource connection and compatibility constraints (P and S)
- Precedence constraints (P and S)
- Spanning constraints (P and S)
- Pegging cardinality constraints (B)
- Maximal inventory constraints (P and S)
- Tank constraints (P and S)
- Shelf life and maturity constraints (all modules)
- Campaign duration constraints (P)
- Secondary resources on setup activities (in some cases planning, in cases scheduling)

Depending on the model data, it may be possible to relax some constraints during optimization.

Another constraint for planning is general *integrality* which, if relaxed, means that all boolean and integer variables are considered to be floating points; a pure linear model results. This relaxation is possible only for planning.

Another constraint for scheduling is Reservoir capacity; this is discussed as the parameter `RESERVOIR_ALLOWED` in *PPO parameters*.

Post processing

Material rebalancing is a post processing option; that is, it can be invoked after the main optimization process of planning, batching, and scheduling occurs. It is available on the **Optimize the scenario** dialog box in the GUI, as well as in the data model via the optimization profile data schema table and API classes.

Material rebalancing is provided for two primary use cases. The first is to correct the balance of semi-finished products by resizing and repegging finished product orders. This is a typical use-case when using problem decomposition, and the semi-finished orders are a little late with respect to the finished product plan.

The second main use case is for *late differentiation*, or adapting the plan of finished products to a change in demand while keeping the same semi-finished product plan.

In either case, there are a set of common guidelines:

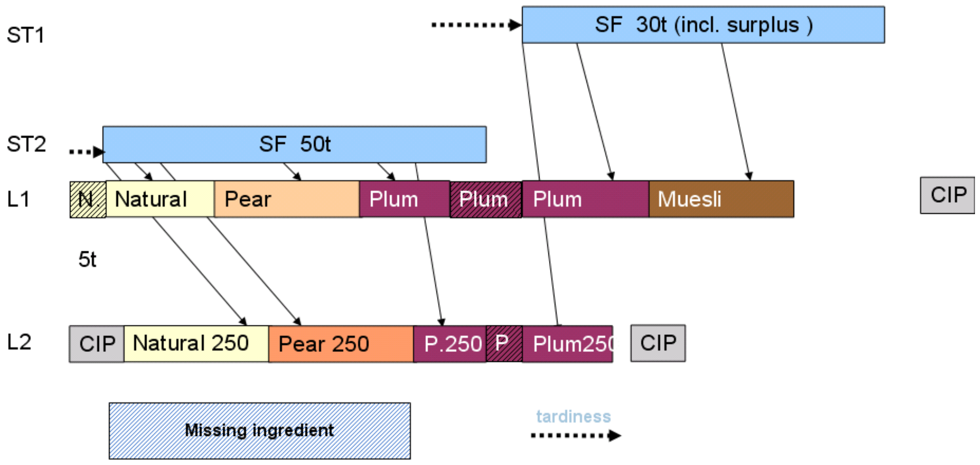
- The semi-finished product schedule is not changed at all;
- Only finished product orders are changed;
- Existing non-firm finished product orders are resized and can be removed if too small;
- Existing finished product orders can be moved within the time bucket that they are scheduled in, but cannot change resource;
- CIPs and fixed orders are not moved;
- Existing finished product orders cannot jump over existing CIPs, breaks, or fixed orders;
- Setup times are managed;
- Sequence is conserved in that there is no appearance of new types of finished products, only a resizing of existing finished products;
- Alternative pegging is taken into account, but a finished product on a previous time bucket cannot be pegged to a semi-finished product of the next time bucket;
- Inventory corridor costs and waste cost are minimized.

Note that there are some limitations upon the recipes of finished products:

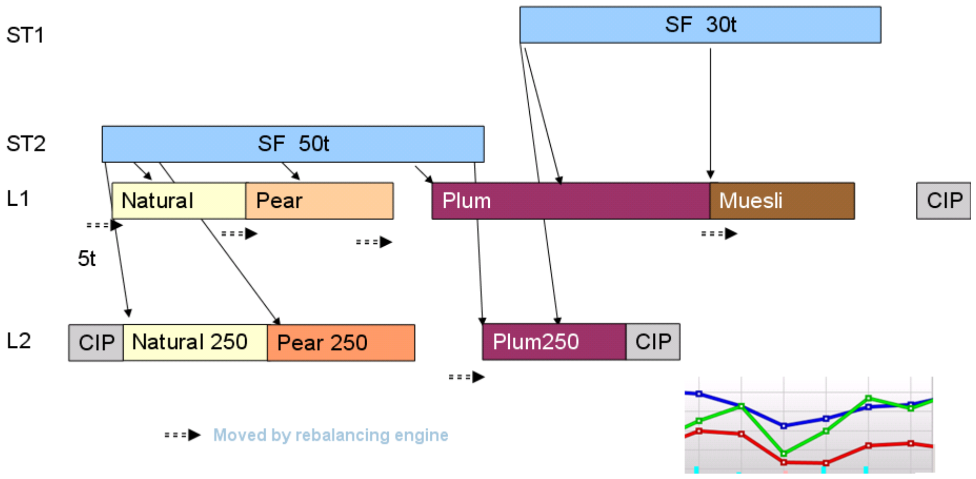
- Recipes for finished products must have a single production activity;
- The processing time must be purely proportional to the batch size;
- Secondary resources are not handled.

Example

This example shows how material rebalancing would be used in the case of correcting the balance of semi-finished products by manipulating the finished product orders. The two semi-finished orders fill two storage tanks (ST1 and ST2) but with tardiness problems.



With material rebalancing, the semi-finished products are balanced again, as shown below. The horizontal arrows show orders moved by rebalancing.



Parameters for optimization

Parameters are available to modify optimization in Plant PowerOps.

PPO parameters

The parameter `modeOfPlanningOnly` is a boolean value (default is false). If the value is true, then the scheduling engine is forced to respect the decisions taken by the planning engine with respect to chosen resources. Note that if super resources have been defined, then the scheduling engine choices are restricted to the subresources of the super resource chosen by the planning. This is a way to limit a possible combinatorial explosion of modes in scheduling.

The parameter `slackOnPlannedStartTime` is an integer value (default is zero if inventory corridors are defined; otherwise `IloMSIntPlusInfinity`). It defines the number of time units of anticipation allowed to the scheduling engine with respect to the start of the time bucket chosen by the planning engine.

The parameter `slackOnPlannedEndTime` expects an integer value (default is `IloMSIntPlusInfinity`). It defines the number of time units of delay allowed to the scheduling engine with respect to the end of the time bucket chosen by the planning engine.

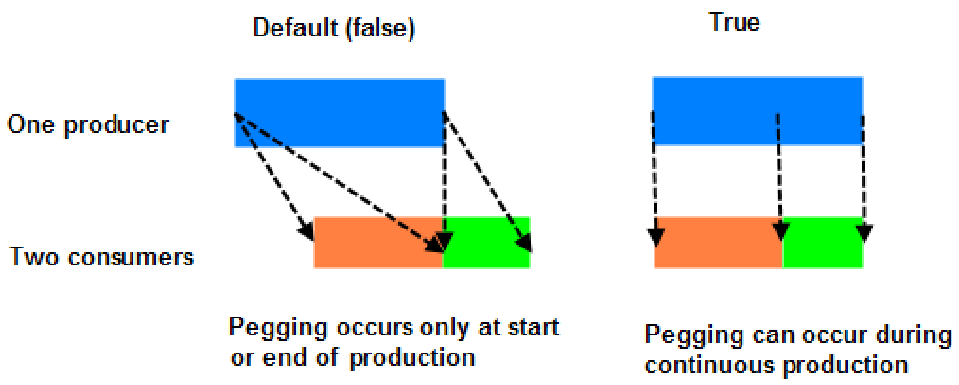
The parameter `RESERVOIR_ALLOWED` (and the constraint **Reservoir Capacity** on **Scheduling advanced options**) is a boolean value that manages constraints on storage unit and inventory capacity. The capacity constraints are implemented through the use of a scheduling structure called the "reservoir." If you set this parameter to false (or relax the constraint on the **Scheduling advanced options**) then you inhibit overflow constraints on storage units (`PPO_STORAGE_UNIT|QUANTITY_MAX`) and on inventories (`PPO_MATERIAL|INVENTORY_CAPACITY`). Relaxing this constraint could result in overflow on storage units and inventory.

Reservoirs are also used to manage inventory excess and deficit costs, inventory min and max cost functions, and used with service levels. In some exceptional cases, the use of the reservoir data structure may be costly in memory or CPU time. To prevent the use of reservoirs, the parameter must be set to false (or relax the constraint) and the inventory excess and deficit weights must be set to zero.

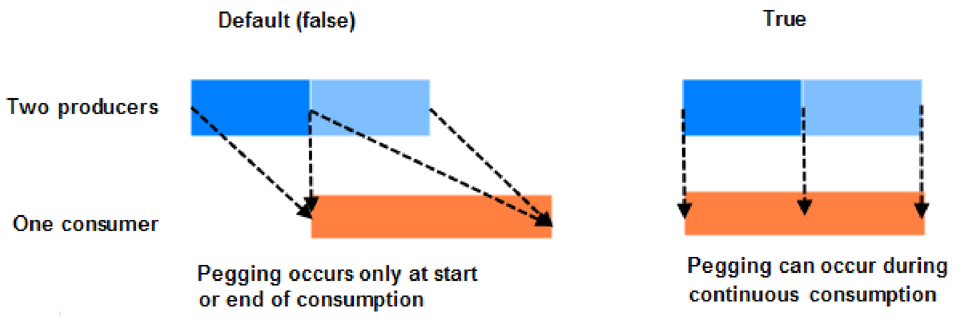
bUseNewSchedulingModelForPeggingArcs

The parameter `bUseNewSchedulingModelForPeggingArcs` permits greater accuracy in the scheduling of pegging arcs with continuous production or consumption. Using this parameter potentially brings improvements in tardiness and nondelivery costs.

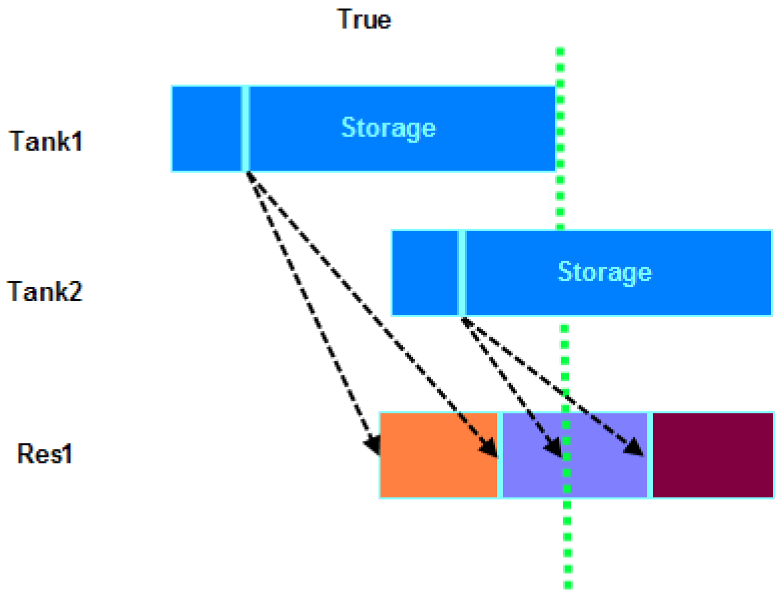
The following diagram represents the 1:N case (one continuous producer and several consumers). The blue graphic represents a single continuous production activity, feeding two consuming activities. The right side illustrates use of the parameter, allowing pegging to occur at times other than the start or end of the continuous production. This may bring the most benefits when the production speed is slower than or equal to the consumption speed.


Default (false)
True
One producer
Two consumers
Pegging occurs only at start or end of production
Pegging can occur during continuous production

In the N:1 case (many producers and one consumer), the corresponding benefit arises: Pegging is allowed to occur at times other than the start or end of the continuous consumption. This may bring the most benefits when the production speed is equal to or greater than the consumption speed.


Default (false)
True
Two producers
One consumer
Pegging occurs only at start or end of consumption
Pegging can occur during continuous consumption

The `bUseNewSchedulingModelForPeggingArcs` parameter also brings benefits in the many-to-many case (N:M) with storage tanks. Previously, N:M relationships with storage activities were not allowed in the scheduling engine (so the scenario for the parameter's default false case is not shown in the following graphic). Using the parameter allows for N:M pegging with tank storage activities as shown in the following diagram.



Multiple storage tanks can be used in a serial manner to meet continuous consumption.

Note that the parameter `bUseNewSchedulingModelForPeggingArcs` affects use of the **Material flow extent** repair mode. When swapping two consumers pegged statically to the same producer, the consuming and producing intervals are recomputed.

CPLEX parameters

Parameters of the type `CPX_PARAM_***` are applicable when solving using the planning engine, as this engine is based on IBM ILOG CPLEX.

CPX_PARAM_EPGAP (numeric, between 0.0 and 1.0): the default value for this parameter is 0.01 (1%). When the value is $E > 0.0$, a planning solution of cost C is known, and a lower bound for the cost of the best (unknown) solution *Cmin* is known, search will stop if the relative difference between *Cmin* and C is less than E. For example, if the value of this parameter is set to 0.01, optimization will stop as soon as it is proven that the cost of the current solution is not more than 1% above the optimum.

Complexity considerations

The response time of the planning engine is a function of several characteristics of the problem.

First, there is the size of the problem, which is mainly a function of:

- Number of buckets * number of recipes
- Number of buckets * number of resources
- Number of buckets * number of materials.

Another complexity issue is the existence of noncontinuous variables, which depends upon:

- The type of setup approximation (adds binary variables)
- The existence of fixed processing times and costs (adds integer variables)
- The use of integral constraints for batch size (adds integer variables)

These variables complicate the problem and may transform it into a complex MILP (Mixed Integer Linear Problem).

Problem decomposition

Several methodologies are available to help you simplify the resolution of a problem model. One method is to use super resources; see *Planning with super resources*. Another method is to decompose the model into submodels which can then be solved separately.

Several schema tables are available to assist in creation of a submodel: PPO_RECIPE_FAMILY, PPO_RECIPE_FAMILY_FILTER, and PPO_SCOPE.

A column in the PPO_OPTIMIZATION_PROFILE called SCOPE_ID can be used to limit the scope of the optimization process, thus performing a natural problem decomposition. There is no need to write code to decompose a problem into a submodel, simply define the necessary recipes and optimization profiles in the database. The resulting scopes will be available on the **Optimize the scenario** dialog box in the PPO GUI.

Advanced decomposition is also available, for use within plug-ins. The classes in the object model are `IloMSRecipeFamily`, `IloMSRecipeFamilyFilter`, and `IloMSScope`. The following API is available to deal with submodels:

- ♦ `masterModel.newScope();`
- ♦ `optimizationProfile.setScope(scope)`
- ♦ `masterModel.newRecipeFamilyFilter(scope)`
- ♦ `filter.addFrozenRecipeFamily(familyX) // freeze any order from recipes of familyX`
- ♦ `filter.addPlannedRecipeFamily(familyY) // plan only with recipes of familyY`
- ♦ `IloMSModel subModel = scope.buildSubModel()`
- ♦ `if (subModel.solve()) scope.transferResults(subModel, masterModel)`

Problem decomposition is discussed in greater detail in *Decomposition framework*.

The basic data model

This section describes the fundamental elements of a data model for Plant PowerOps. This information helps you populate a problem model in the context of a given application.

In this section

Overview of data modeling for PPO
Includes a definition of a simple data model, description of costs and revenue, and methods of modeling data.

Global model information
Describes overall information relating to the model environment, time, and horizons.

Materials
Describes the modeling of materials. In PPO, the term *material* includes products, intermediates, and raw materials.

Resources
Resources include machines, tools, workers, vehicles or equipment.

Demands and due dates
Describes modeling customer orders and forecasts.

Recipes
Recipes are a set of activities with alternative modes.

Production orders and material flow arcs
Together these items comprise the batching solution.

Procurements

Procurements represent materials procured from outside the plant, and are also used to represent initial stock.

Firm or fixed information

Firm information is used to model certain orders and quantities that cannot be changed.

Overview of data modeling for PPO

Includes a definition of a simple data model, description of costs and revenue, and methods of modeling data.

In this section

Introduction to a data model, master data, and transactional data
A list of what is included in a simple data model, master data, and transactional data.

Costs and revenue
Describes costs and revenue.

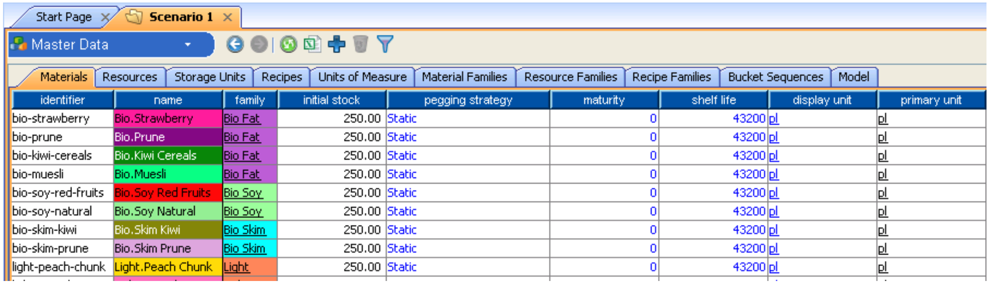
Methods of modeling data
Describes the ways available to you to create models of your manufacturing data.

Introduction to a data model, master data, and transactional data

A basic data model for PPO can include some or all of the following:

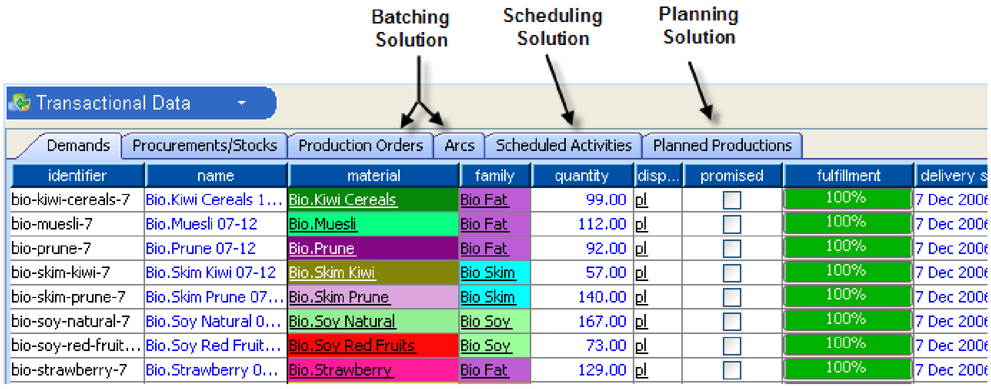
- Overall *model* information, such as time unit, start time, and *Optimization profile*;
- A list of *materials* that represents the raw materials, manufacturing intermediates, and final products;
- A set of *resources*—objects such as machines, workers, and vehicles, which add value to a product or service in its creation, production, or delivery;
- *Demands* and *due dates* that represent customer orders or forecasts for a given amount of material at a given time;
- A set of *recipes* that describe the process by which resources are used to transform the raw materials into final products;
- *Production orders* that implement recipes, and *material flow arcs* that describe the resulting flow of material through the process and plant;
- A set of *procurements* representing initial stock quantities or material obtained from outside the plant;
- *Firm information* representing parts of a schedule that are not changeable, for example because certain production operations are already in process.

This model information is grouped into one of two categories: *master* data or *transactional* data. Master data represents the base information for your manufacturing problem and facility, and includes the model, materials, resources, and recipes. This information does not change often nor does it include solution data (data optimized by PPO). The **Master Data** view in the PPO GUI is shown below.



identifier	name	family	initial stock	pegging strategy	maturity	shelf life	display unit	primary unit
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	0	43200	pl	pl
bio-prune	Bio.Prune	Bio Fat	250.00	Static	0	43200	pl	pl
bio-kiwi-cereals	Bio.Kiwi Cereals	Bio Fat	250.00	Static	0	43200	pl	pl
bio-muesli	Bio.Muesli	Bio Fat	250.00	Static	0	43200	pl	pl
bio-soy-red-fruits	Bio.Soy Red Fruits	Bio Soy	250.00	Static	0	43200	pl	pl
bio-soy-natural	Bio.Soy Natural	Bio Soy	250.00	Static	0	43200	pl	pl
bio-skim-kiwi	Bio.Skim Kiwi	Bio Skim	250.00	Static	0	43200	pl	pl
bio-skim-prune	Bio.Skim Prune	Bio Skim	250.00	Static	0	43200	pl	pl
light-peach-chunk	Light.Peach Chunk	Light	250.00	Static	0	43200	pl	pl

Transactional data, on the other hand, is subject to frequent change (perhaps on a daily basis), and includes demands, procurements, production orders, and firm information. Transactional data can contain solution data, as shown in the following image.



identifier	name	material	family	quantity	disp...	promised	fulfillment	delivery s
bio-kiwi-cereals-7	Bio.Kiwi Cereals 1...	Bio.Kiwi Cereals	Bio Fat	99.00	pl	☐	100%	7 Dec 200
bio-muesli-7	Bio.Muesli 07-12	Bio.Muesli	Bio Fat	112.00	pl	☐	100%	7 Dec 200
bio-prune-7	Bio.Prune 07-12	Bio.Prune	Bio Fat	92.00	pl	☐	100%	7 Dec 200
bio-skim-kiwi-7	Bio.Skim Kiwi 07-12	Bio.Skim Kiwi	Bio Skim	57.00	pl	☐	100%	7 Dec 200
bio-skim-prune-7	Bio.Skim Prune 07...	Bio.Skim Prune	Bio Skim	140.00	pl	☐	100%	7 Dec 200
bio-soy-natural-7	Bio.Soy Natural 0...	Bio.Soy Natural	Bio Soy	167.00	pl	☐	100%	7 Dec 200
bio-soy-red-fruit...	Bio.Soy Red Fruit...	Bio.Soy Red Fruits	Bio Soy	73.00	pl	☐	100%	7 Dec 200
bio-strawberry-7	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	129.00	pl	☐	100%	7 Dec 200

Vital components of any manufacturing plan are costs and revenue. These items are not separate modeling objects, but rather are elements of master and transactional data objects. For example, inventory costs are attached to material objects, whereas nondelivery costs and revenue are attached to demand. For more information see *Costs and revenue*.

Costs and revenue

Costs can be organized into three categories.

Manufacturing costs:

- Processing costs (also called mode costs); recipes and modes can be used to manufacture materials with differing methods and costs
- Resource setup costs
- Resource capacity costs; several levels of capacity may be available at different costs for each resource over each bucket
- Resource cleanup costs

Holding costs:

- Inventory costs

Customer “dissatisfaction” costs:

- Earliness and tardiness costs (delivery)
- Inventory deficit costs
- **Nondelivery costs** (also called unsatisfied demand)

Nondelivery costs on demands are mandatory in order to provide an incentive for optimization to fulfill those demands. An infinite nondelivery cost is equivalent to a mandatory demand. *Revenue* is essentially the opposite result; it represents the gain achieved by satisfying customer demand, and provides additional incentive to produce final products.

All costs and revenue can be weighted by *Optimization criteria* contained in the current optimization profile. The overall solution objective of optimization is to minimize the weighted costs while maximizing revenue. In the PPO GUI, you may notice an additional component of the global objective called *perturbation*. This is a technical cost that the planning engine uses to break the symmetries of the problem

Methods of modeling data

Your manufacturing plant data can be created and input to PPO in different ways:

- Use features of the PPO GUI, including the new model wizard, data editors, and import and export to Microsoft® Excel® . The new model wizard is available from the **File** menu of the GUI, and assists you in creating the base information for your model. On the **Master Data** view, you can edit a lot of data directly in the tables (blue fields), or access data editors via the contextual menu (right-click), or click the plus icon on the view's toolbar to add a new data row. These editors allow you to add or edit materials, resources, recipes, and more. The **Plant Layout** view allows you to add resources and visually organize the layout. Data exchange with Excel is useful because it allows you to do large global data changes, which is particularly useful in the case of transactional data and when defining a setup matrix.

- Create a file in either comma-separated values (*.csv files) or in a Microsoft Access® database (*.mdb files) format which includes tables of the model data. The *Data Schema* is the defining document for creating a data model, detailing all the available tables, fields, default values and ranges. Once created, the data file can be solved by loading into the GUI, or with C++ or Java™ programs if you prefer. PPO documentation provides examples and tutorials to show you how to analyze your problem data and model it.

- Use the Plant PowerOps C++ application programming interface (API) to describe the constraints and criteria of your problem. The C++ API reference manual is located in the *Reference Documentation* section.

- Use the Plant PowerOps Java API to describe the constraints and criteria of your problem. The Java API reference manual is located in the *Reference Documentation* section.

- PPO also accepts input from Oracle® and SQL Server® databases. This is described in *Database usage and connectivity*.

Consider which modeling method best fits the situation at your facility. For example, building an entire model starting with empty csv or mdb files is not likely to be the quickest method. Spreadsheet and .mdb files are a convenient way to store a scenario, but a poor solution for creating a data repository or for integration of a real project. Prototyping your model will be achieved more efficiently through use of the PPO GUI features mentioned above, the load and save file features of the GUI, and by editing or loading data files from outside PPO as desired. Integration with a live project could be achieved using a true database or using the API to map an intermediate model to PPO.

Examples and descriptions in the documentation tend to rely on the relational model of the data schema. The schema tables makes it easy to illuminate concepts in a step by step manner, and this fits the needs and comprehension of many users. However, almost all PPO features are readily available across all model formats. Note that regardless of the technology used, PPO uses a single object model for representation of both the planning and scheduling problems.

Global model information

Describes overall information relating to the model environment, time, and horizons.

In this section

The model environment
An instance of the model provides the environment for the problem.

Time units and time buckets
How time units and time buckets are used in PPO.

Planning horizon, scheduling horizon, and end max
Describes how these model elements interact.

The model environment

The model contains information such as:

♦ A *name* for the model.

♦ The current optimization profile used by default when solving. The profile includes solving parameters and describes the purpose for which you use PPO: Pure planning, pure scheduling, or integrated planning and scheduling.

♦ The time zone for displaying time in the local time zone of the plant. The PPO GUI respects the daylight savings time.

♦ A *date origin* that corresponds to time zero in the model, with which all relative times are interpreted. The given date is provided in the defined time zone or in UTC, depending upon your method of data modeling.

♦ A minimal start time (*start min*) for all the activities in the model, that represents the earliest possible time that an activity can start on a resource. This defines a date for the initial state of the problem scenario. Any procurement received before this date is considered to be initial stock.

♦ A maximal end time (*end max*) that represents the latest end time available to activities in the model.

♦ The *time unit* in seconds, used to express durations in your model. For example, use a time unit of 60 if you wish to provide data and see scheduling results in minutes (or 1 in seconds, 3600 to work in hours, and so on). The start min, end max, and other temporal data is typically expressed in the declared time unit. Your choice of time unit depends on the desired detailed scheduling precision; the scheduling module will check constraints based on the time unit.

♦ The *bucket sequence* used in the model, which defines how time buckets are defined.

These items are all elements of the model data object, represented by the `PPO_MODEL` table in the relational model, or the `IloMSModel` class in the API. However, this data closely interacts with elements from the optimization profile, discussed in *Time units and time buckets* and *Planning horizon, scheduling horizon, and end max*.

See also:

♦ The `PPO_MODEL`, `PPO_OPTIMIZATION_PROFILE`, `PPO_BUCKET`, `PPO_BUCKET_SEQUENCE`, and `PPO_BUCKET_TEMPLATE` tables in the data schema.

♦ The classes `IloMSModel`, `IloMSOptimizationProfile`, `IloMSBucket`, `IloMSBucketSequence`, and `IloMSBucketTemplate` in the API documentation.

♦ *Optimization profile*.

♦ An example of the `PPO_MODEL` table in *Step 3 Create the environment* of *Modeling a simple problem: A “bottleneck” resource*.

♦ In the PPO GUI on the **Master Data** view, overall model information appears on the **Model** tab. The **New Model** wizard, available from the **File** menu, is an easy way to create global model information.

Time units and time buckets

Although the scheduling module checks constraints at the time unit, the planning engine generally requires a larger time boundary for long term planning; this requirement is fulfilled by the *time bucket*. While creating the production plan, the planning engine checks constraints at time bucket boundaries. PPO will create the time buckets automatically if you do not model them, but you may wish to define your own. For example, you can define two time bucket sequences so that a daily time bucket is used for optimization, while the GUI reports weekly planning data. See *Defining time buckets* for more information.

Planning horizon, scheduling horizon, and end max

The *horizons* for the model are declared in the optimization profile, and they define the end times for production planning and for detailed scheduling. Separate horizons are available in PPO since it is common to make short-term detailed schedules and long-term production plans.

The *end max* is declared in the model object, and it defines a common latest end time associated with all the activities of the model.

The horizons and end max use positive infinity as a default value, but as these elements can be set independently it is important to understand how they interact. For example, if you do not set the planning and scheduling horizons, but define a model end max, then the horizons are effectively limited by the model end max. If the planning module has been too “optimistic” with production, this will lead to an infeasible problem.

A good practice is to specify the horizons and end max such that:

Scheduling horizon **<** *model end max*

Scheduling horizon **<** *planning horizon*

Compare these relations in terms of time bucket values (rather than time unit) in order to ensure significant differences.

There is another consideration to be noted in regards to solution display in the GUI. For integrated planning and scheduling problems, if the scheduling horizon is less than the planning horizon, then for time periods before the scheduling horizon the solution data is based on the scheduling solution. For time periods after the scheduling horizon, the planning solution data (planned productions and planned deliveries) are displayed in the appropriate views of the GUI.

Materials

Materials in PPO are used to represent raw materials, intermediate products, and final products. Every final product, whether ordered by a customer or destined for inventory, must be modeled. Generally, you only need to model the raw materials or intermediate products for which some supply issue or shelf life constraint may exist, or for which a flow discontinuity exists (batch splitting or merging).

A material may have unique characteristics of size, style, and inventory, corresponding to unique stock-keeping units (SKU). For example, consider two models of white shirts that are stored in the same warehouse but differ in size or style; each model of shirt would be a different PPO material (as a record of the `PPO_MATERIAL` table or an instance of `IloMSMaterial`). Likewise, one type of shirt with an inventory spread across two warehouses would likely be modeled as two materials. In this latter case, the two materials could be grouped together into one or more *Material families*.

It's possible to define more than one unit of measurement per material. The primary unit is used for all computations: Production, consumption, inventory, and so forth. Secondary units can be declared, each of which has a conversion factor that relates it to the one primary unit. Then you can select one of these secondary units as the display unit for use in the GUI. Also, several materials may share the same unit with different conversions.

Crucial concerns with materials are inventory levels, storage, maturity, and shelf life. You can specify a maximal inventory capacity for a material in either a fixed-storage location manner or in an as-available location. These subjects are discussed in greater detail in *Managing inventory stock* and *Maturity, shelf life, waste recipes*.

You can specify an initial quantity of material that is available independently of any material flow arc or manufacturing operation. This means the actual quantity of material available at time `start_min` is equal to the initial quantity, plus the quantities brought or removed by past material flow arcs. Note that you cannot model shelf life, maturity constraints, or storage information for an initial material quantity.

Material families

A material can belong to one or more material families, which allows for customized data organization and tracking. For example, you can view production and inventory reports in the GUI organized by individual materials or by material family. Use the **family type** and **aggregate materials** selections on the **Parameters** window to control views such as **Stock Event** and **Stock Coverage**.

The material family provides a useful planning constraint; you can limit the maximum number of different materials of a family produced in a time bucket. The time buckets used on this constraint do not have to be the same size as those used for planning optimization. For example, you could produce no more than three different types of organic vegetarian soups per week while planning all soup production on daily buckets.

See also:

- ♦ `PPO_MATERIAL`, `PPO_MATERIAL_FAMILY`, `PPO_MATERIAL_FAMILY_CARD_CONSTRAINT`, and `PPO_MATERIAL_SECONDARY_UNIT` tables in the data schema.
- ♦ Classes `IloMSMaterial`, `IloMSMaterialFamily`, and `IloMSMaterialFamilyCardinalityConstraint` in the API documentation.

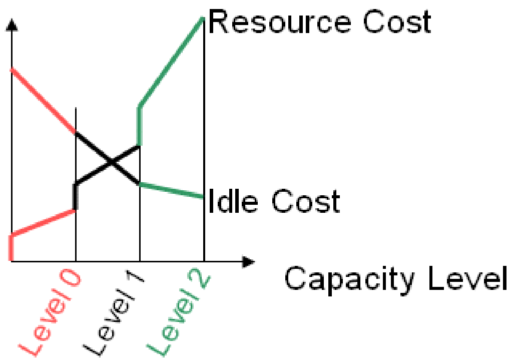
- *Managing inventory stock*
- *Maturity, shelf life, waste recipes*
- Material production and consumption is discussed in *Recipes*.
- An example of the `PPO_MATERIAL` table in *Step 5 Define the materials* of *Modeling a simple problem: A "bottleneck" resource*.
- In the PPO GUI on the **Master Data** view, materials information appears on the **Materials**, **Material Families**, and **Units of Measure** tabs, and may be available for editing.

Resources

Resources include items such as machines, tools, workers, vehicles or equipment which add value to a product or service in its creation, production, or delivery. By default, a resource is available for use throughout the time horizon of the problem, but that can be modified for cleaning, maintenance, or other reasons. Raw materials and intermediate products are not considered resources in PPO; use *Materials* to model those items.

When several resources are roughly equivalent in terms of function, capability and availability, and do not require any setup or cleanup, it is recommended that the set of resources be modeled as a single resource. Then derive the capacity of the single resource based on the capacity of the group; for example, if you have eight workers with interchangeable skills, model this as a single worker resource of capacity eight.

You can define resource usage and idle costs with a *resource capacity cost function*. The cost for using a resource often increases with the capacity used; this resource capacity cost can be defined as a piecewise linear function or as a stepwise linear function. Idle cost can be defined as a piecewise linear function.

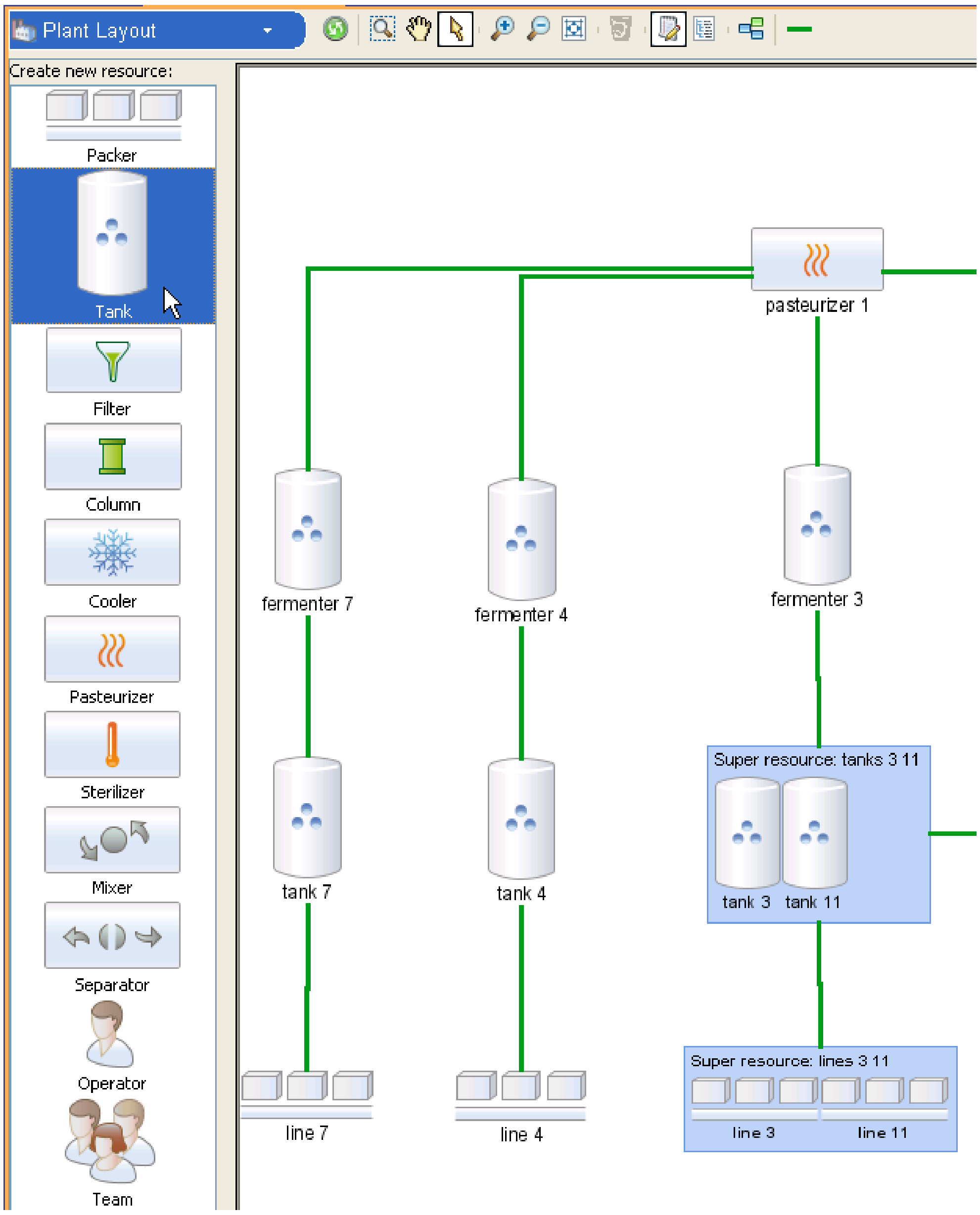


Resource costs and idle costs enable you to more evenly distribute resource usage in production planning. These planning-related costs are not to be confused with processing costs, which are applied to resource use for production activities (in defined modes). Processing costs are discussed further in *Recipes*.

Several factors can affect resource availability: Setup requirements, cleaning requirements, and regularly scheduled events (like worker breaks) which can be modeled with *calendars*. For more information on these subjects see *Setup times and costs*, *Cleanups*, and *Calendars*.

Several options are available regarding resource capacity. One is the *required capacity*, which defines the amount of capacity of the resource that is required to execute an activity in a certain mode (discussed in *Recipes*). Another option is the *planning capacity reduction factor*, which allows you to limit the bucket capacity of the planning engine so that production planning is not too optimistic regarding resource availability with respect to real scheduling constraints.

Not surprisingly, resources play a central role in several views in the PPO GUI: The **Gantt Diagram**, **Workload Table**, **Planning Sheet**, and **Calendars** all display aspects of resource usage. The **Plant Layout** view provides an interface for easily adding, removing, and editing resources and resource connections.



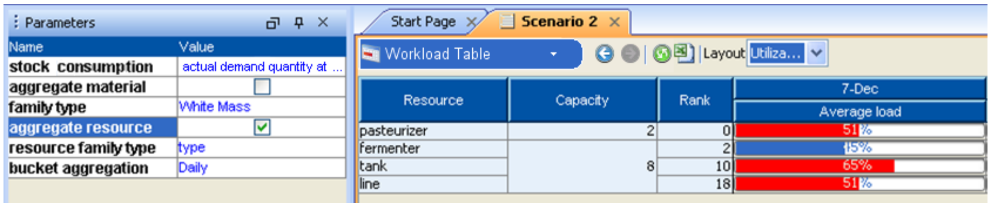
Resource organization

As mentioned previously, it is recommended that whenever possible you should model similar resources together into one resource and assign an appropriate capacity for the group. A related method of grouping resources is to use *super resources*, which group similar resources together to allow the planning engine to consider the grouped capacity of all the resources. The purpose of using super resources is to lighten the planning problem; the scheduling engine utilizes the individual resource entities, not the super resource. For more information see *Planning with super resources*.

Since activities may require the use of more than one resource, you can define one *primary* and multiple *secondary* resources in a mode, along with the capacity required from each. Note however that a resource cannot be used as the primary resource by one activity and as a secondary resource by other activities. Also, a secondary resources cannot support setups and cleanups.

Note that you can define a *resource connection* that models a physical connection between resources so that you can enforce the usage of two connected resources as primary resources of two activities. The point of this constraint is to enforce that two given activities will be executed in compatible modes. It is permitted to model connections between super resources and individual resources. In addition to connecting the resources you must model a compatibility constraint of type *ConnectedPrimaryResource* on the two activities of the recipe.

Another modeling structure is the *resource family*. The point of the resource family is to permit organization of data in the PPO GUI for better visibility. For example, the following image shows the **Workload Table** with an **Average load** graph displaying usage rates of various resources. The resources have been grouped, so that the planner can get a quick overall view of how busy all the fermenters or lines are on that day. The resources have been grouped according to their family on the **Parameters** window, using the **aggregate resource** check box and **resource family type** selection. The selections on the **Parameters** window has similar effects on other views such as the **Gantt Diagram**. A resource may be a member of several families, provided that the families are of different types.



Name	Value
stock consumption	actual demand quantity at ...
aggregate material	☐
family type	White Mass
aggregate resource	✓
resource family type	type
bucket aggregation	Daily

Resource	Capacity	Rank	7-Dec
			Average load
pasteurizer	2	0	51%
fermenter		2	45%
tank	8	10	65%
line		18	51%

See also:

- PPO_RESOURCE, PPO_RESOURCE_CAPACITY_COST, PPO_RESOURCE_FAMILY, PPO_SECONDARY_RESOURCE, PPO_RESOURCE_HIERARCHY, and PPO_PROD_PROD_COMPAT_PROTO tables in the data schema.
- Classes IloMSResource, IloMSResourceCapacityCostFunction, IloMSResourceFamily, and IloMSActivityCompatibilityConstraint in the API documentation.
- *Planning with super resources*
- *Resource validity periods*

- *Calendars*
- An example of the `PPO_RESOURCE` table at *Step 4 Define the resource*, of *Modeling a simple problem: A “bottleneck” resource*
- In the PPO GUI on the **Master Data** view, resource and resource family information appears on the **Resources** and **Resource Families** tabs, and may be available for editing.

Demands and due dates

A demand object in PPO represents a forecast or customer order request for a certain quantity of finished products. A time window for delivery can be specified, with an optional preferred due date. The due date defines the optimal time for delivery of demanded material, and is used to compute any earliness and tardiness costs. It is highly recommended to narrow the delivery time window in order to limit the complexity of the planning problem.

It is mandatory to apply a nondelivery cost to a demand in order to provide an incentive to the planning engine to meet that demand. An infinite nondelivery cost creates a mandatory demand. You can also apply revenue to satisfaction of demand. Nondelivery cost and revenue are calculated on a “per unit of demand” basis. The model checker of PPO issues a message if both revenue and nondelivery cost are undefined or if both are set to 0.

Plant PowerOps places a higher priority on satisfying demands with high revenues and high nondelivery variable costs. Typically, customer demands have higher nondelivery costs than stock orders and estimated forecasts. It is possible to set the nondelivery cost to infinity, thereby making a delivery mandatory; however, do this only when you are certain that the demand can be met. Otherwise, PPO may conclude that your problem has no solution.

A demand may be delivered in several subdeliveries. Hence, part may be tardy and part may be early; part may be delivered, and part may not. For each unit of material that is delivered, the revenue is obtained. For each unit of material that is not delivered, the nondelivery cost is incurred. For example, suppose *D* is a demand for 100 units of material *M* that provides a revenue per unit of 100 and a nondelivery cost per unit of 50. Suppose that 70 units are delivered; then the total revenue for this demand is 7000, and the total nondelivery cost 1500.

Each subdelivery can be considered as an activity of duration at least 1. For each of these subdeliveries, the inventory of the demanded material is decremented at the subdelivery end time; tardiness is incurred if the subdelivery end time strictly exceeds the due date; earliness is incurred if the subdelivery end time strictly precedes the due date. If the earliest delivery start time is defined, none of the subdeliveries can start before this time. If the latest delivery end time is defined, none of the subdeliveries can end after this time. Note that the earliest delivery start time must be strictly lower than the latest delivery end time, otherwise it is not possible to deliver.

To define the optimal time, four costs may be defined on the due date:

Earliness fixed cost: A fixed cost to be paid per unit of material if the demand is delivered early, regardless of how much it is early. In the planning model, this cost will be incurred if the material is delivered in any bucket that precedes the due date.

Earliness variable cost: A cost to be paid per unit of material and per unit of time that the delivery is early. In the planning model, this cost will be multiplied by the difference between the due date and the end time of the bucket in which the material is delivered.

Tardiness fixed cost: A fixed cost to be paid per unit of material if the demand is delivered late, regardless of how much it is late. In the planning model, this cost will be incurred if the material is delivered in any bucket that follows the due date.

Tardiness variable cost: A cost per unit of material and per unit of time that the delivery is late. In the planning model, this cost will be multiplied by the difference between the start time of the bucket in which the material is delivered and the due date.

There are several optional requirements that you can place on a demand. You can specify the storage unit from which the material must be delivered to meet this demand. You can require a minimal remaining shelf life of any material delivered to meet this demand. You can also specify a maximum number of arcs associated with this demand; that is, ensure the number of deliveries to meet this demand does not exceed a certain value.

See also:

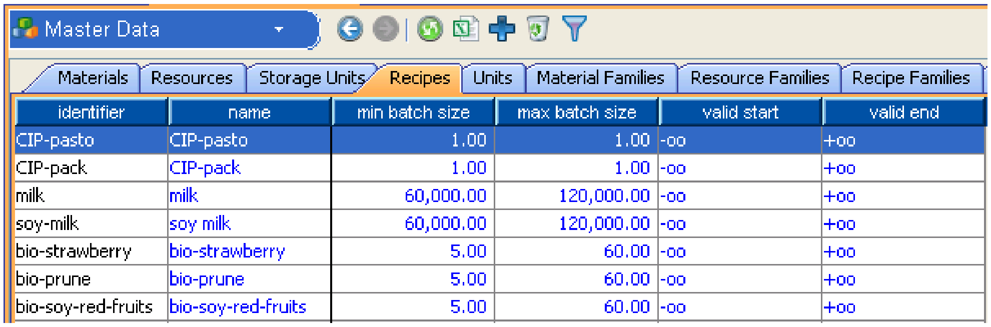
- `PPO_DEMAN`D and `PPO_DUE_DATE` tables in the data schema.
- Classes `IloMSDemand` and `IloMSDueDate` in the API documentation.
- An example of the `PPO_DEMAND` table at *Step 10 Define the demand* of *Modeling a simple problem: A "bottleneck" resource*.
- An example of the `PPO_DUE_DATE` table at *Step 11 Define due dates for the demands* of *Modeling a simple problem: A "bottleneck" resource*.
- In the PPO GUI on the **Transactional Data** view, demand and due date information appears on the **Demands** tab, and may be available for editing.

Recipes

Generally, recipes model the processes at your plant facility. This includes all production processes as well as the supporting, non-production processes: Cleanup recipes are used to clean resources, transport recipes transfer material from one storage unit (or location) to another, and waste recipes are used to discard obsolete material.

Recipes are essentially a set of activities; each of which can be performed in one or several alternate production modes, and each may produce or consume materials. The activities can be affected by various constraints, and may have differing resource requirements depending upon the production mode. Recipes are implemented by *Production orders and material flow arcs* in order to meet demand.

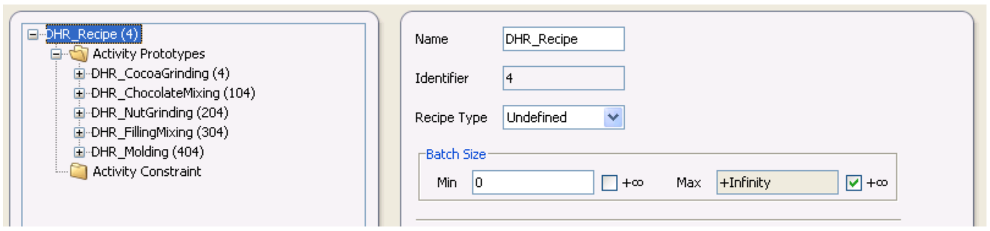
The following image shows the **Recipes** tab on the **Master Data** view in the PPO GUI. It shows the name of the activity, and the **batch size** which is a multiplication factor on the recipe. The batch size is used by production orders to compute the quantities of materials produced or consumed, and also adjusts the processing times and costs of activities (if variable processing times and costs have been specified). The **valid start** and **valid end** times allow you to specify when the recipe can be used; see *Resource validity periods* for more information.



identifier	name	min batch size	max batch size	valid start	valid end
CIP-pasto	CIP-pasto	1.00	1.00	-oo	+oo
CIP-pack	CIP-pack	1.00	1.00	-oo	+oo
milk	milk	60,000.00	120,000.00	-oo	+oo
soy-milk	soy milk	60,000.00	120,000.00	-oo	+oo
bio-strawberry	bio-strawberry	5.00	60.00	-oo	+oo
bio-prune	bio-prune	5.00	60.00	-oo	+oo
bio-soy-red-fruits	bio-soy-red-fruits	5.00	60.00	-oo	+oo

As with materials and resources, recipes can belong to families, with corresponding information available on the **Recipe Families** tab shown above **Master Data** view. Recipe families are an important method of problem decomposition and partial resolves; recipe families are a way to define submodels and to perform a partial database load of the data. Loading a submodel from a database is discussed in *Opening a database in PPO*.

If you right-click a recipe in the **Recipes** table, a contextual menu appears with an **Edit** selection; this selection displays the **Recipe Editor** window, part of which is shown here.



- DHR_Recipe (4)
  - Activity Prototypes
    - DHR_CocoaGrinding (4)
    - DHR_ChocolateMixing (104)
    - DHR_NutGrinding (204)
    - DHR_FillingMixing (304)
    - DHR_Molding (404)
  - Activity Constraint

Name: DHR_Recipe

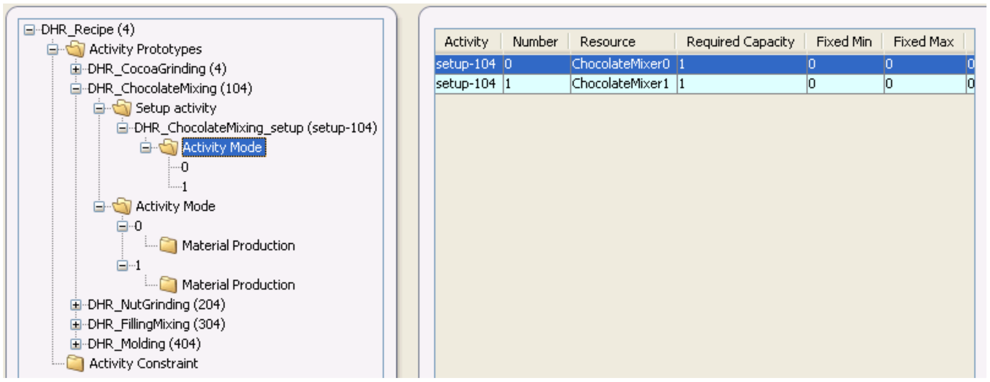
Identifier: 4

Recipe Type: Undefined

Batch Size

Min: 0 [ ] +∞ Max: +Infinity [✓] +∞

In this example, the editor window is open to a recipe called DHR_Recipe, which consists of five activity prototypes which model processes like cocoa grinding and chocolate mixing. The next image shows the editor window with one of these activity prototypes open, revealing that this prototype contains a setup activity with two possible modes (both of which prepare the resource for use) and two material production modes that produce mixed chocolate. These terms (modes, prototypes, constraints, and material production) are discussed further in the next sections. Generally speaking, it can be said that all of these objects together to create the recipe.



Activity	Number	Resource	Required Capacity	Fixed Min	Fixed Max	
setup-104	0	ChocolateMixer0	1	0	0	0
setup-104	1	ChocolateMixer1	1	0	0	0

Activities

There are three types of activities in PPO: Production, cleanup, and setup activities. A cleanup activity is considered to be a special type of production activity. This section describes production activities; setup activities are considered in greater detail in *Setup times and costs*.

Generally, an activity has three phases in PPO: As prototype, generated instance, and scheduled. The activity prototypes belongs to recipes, and the generated instances belong to production orders. An activity prototype is used in production planning as the abstract model of an actual activity that may be performed many times in a plant. The activity prototypes are used as the model or template to create the generated activities that are necessary to fulfill the production plan. Detailed scheduling then assigns each of the generated activities to precise start and end dates, and the results can be viewed in the **Scheduled Activities** data table in the GUI. The generated activities are cloned from the prototypes at the end of the batching process. just before the detailed scheduling module is called.

In most cases, your modeling responsibility is to create the activity prototypes, and allow PPO to generate the activities and detailed schedule. Using the data schema, this means modeling using the `PPO_ACTIVITY_PROTO` table, and PPO creates the generated activities in the `PPO_ACTIVITY` table; whereas when using the C++ or Java API a single class `IloMSActivity` is used.

When modeling the activity prototype, you define the recipe to which the activity belongs, and can assign a color and name to appear in the **Gantt Diagram**. Some production activities cannot be performed unless the resource is “ready” or in a certain setup state (for example, blue paint in a reservoir); see *Setup times and costs* for more information. You can specify the performed status of an activity as *Performed*, *Unperformed*, or *PerformedOrUnperformed*; the last status means that the engine has the option of relaxing the capacity constraint (as if the activity were to possibly be outsourced, so no longer under capacity limitations of modeled resources).

You can specify if the activity is a cleaning activity, that is, if it is used to clean a resource. Resource cleaning is described in *Cleanups*.

Activities can have due dates, with associated earliness and tardiness costs. As examples, the earliness fixed cost is a fixed price to be paid if the activity is early. For example, it represents the cost of entering a final product into a warehouse, rather than shipping it directly to the customer. The earliness variable cost is the price to be paid (in addition to the earliness fixed cost) per time unit that the activity is early. For example, it represents the cost of using additional space in a warehouse for an early final product.

You can create an *activity chain* which is a sequence of production activities that are performed successively on the same primary resource with the same setup state. This could be useful in certain cases, such as where each individual activity requires different scarce secondary resources that need to be taken into account to build a realistic schedule. Another example is if some of the individual activities cannot be interrupted by "breaks" (such as lunch breaks), but breaks are allowed within the activity chain (that is, between the individual activities).

You can apply various constraints to or between activities; see *Precedence constraints* for more information.

Modes

A mode is a way of performing an activity; for example, which resource is used, how long the activity takes, how much it costs to perform. Depending on the resource used, an activity might have a longer or a shorter processing time or a lower or higher cost, and you can express this through modes.

The processing time consists of two parts: a fixed part independent of the batch size, and a variable part that depends upon the batch size. The effective processing time of a generated activity (for a given production order with a given batch size) is computed as follows: Effective_Processing_Time = Fixed_Processing_Time(prototype) + Variable_Processing_Time(prototype) * Batch_Size. Note that the prototype Fixed_Processing_Time may vary between a given Fixed_Processing_Time_Minimum and a given Fixed_Processing_Time_maximum. Processing costs also have a fixed part and a variable part. The processing cost of a generated activity is: Effective_Processing_Cost = Fixed_Cost(prototype) + Variable_Cost(prototype) * Batch_Size. The total processing cost (a criterion) is the sum, over all activities, of the cost of the selected modes.

You can specify a minimal and maximal batch size for the mode, a maximal break duration for the activity in this mode, and whether or not an activity in this mode can overlap a work shift change.

Note that you can also specify a minimal or maximal batch size on recipes. For example, two modes which use tanks of different maximum batch sizes may be available to an activity; this can be modeled with one recipe, that restricts the batch size of the production orders implementing the recipe.

Modes are one of the two data elements that you can associate with a calendar (resources being the other). See *Calendars* for more information.

Material production

The material production data object specifies which activities and modes produce and consume materials, in what quantity, and to/from which storage units. Material consumption is defined as a negative material production. Production and consumption can be discrete or continuous. Discrete production means that the material is not available before the end of the activity, while continuous production means that the material becomes available as the activity executes, that is, the first units of material are produced as soon as the activity begins and the last units are available only as the activity ends.

An activity may produce or consume multiple materials. Within each recipe, a material can only be produced or consumed by one activity; the activity may have multiple modes to produce the material but all must produce the same quantity. No other activities of that recipe may produce that same material but they may produce other materials.

You can set a minimum or maximum material ratio used for blending purposes; that is you can specify that a certain material represents 25% (for example) of all production or consumption. Also you can set the maximum number of material flow arcs associated with this material production in a production order.

See also:

- ♦ Tables in the *Recipes, Activites, and Modes* section of the data schema: `PPO_RECIPE`, `PPO_ACTIVITY_PROTO`, `PPO_MODE_PROTO`, and `PPO_MATERIAL_PRODUCTION_PROTO`.
- ♦ The `IloMSRecipe`, `IloMSActivity`, `IloMSMode`, and `IloMSMaterialProduction` classes in the API documentation.
- ♦ An example of the data tables `PPO_RECIPE`, `PPO_ACTIVITY_PROTO`, `PPO_MODE_PROTO`, and `PPO_MATERIAL_PRODUCTION_PROTO` in *Step 6 Define the recipe* through *Step 9 Define the material production* of *Modeling a simple problem: A “bottleneck” resource*.
- ♦ *Activity constraints, validity periods, and names*.

Production orders and material flow arcs

A production order implements a recipe to (for example) meet demand, storage requirements, transportation needs, or cleanup requirements. Material flow arcs represent the flow of material through the plant, for example from production order to demand or to another production order. Together, production orders and material flow arcs comprise the batching solution from optimization, and are viewable on the **Transactional Data** view in the GUI as shown in *Optimization solution in the PPO GUI*.

While in the general case *recipes* represent the *prototypes* of the manufacturing process, the production orders represent the actual *instances* that are necessary to carry out the manufacturing process. This means that when using spreadsheet or database files, one must distinguish between the recipe tables (such as `PPO_ACTIVITY_PROTO`) and the production order tables (such as `PPO_ACTIVITY`). In the object model, these items are typically accessed via the same classes.

Production orders use the *batch size* as a multiplication factor on the recipe, as needed. The batch size is used to compute the quantities of materials produced or consumed, and also to adjust the processing times and costs of activities (if variable processing times and costs have been specified on the modes of the activity prototypes of the recipe).

Material flow arcs specify the quantity of material that flows between stock, procurements, production orders, and customer demands. In general, such a flow of material induces a temporal constraint (precedence) between the corresponding nodes. As examples, a production-to-demand arc means that some material produced by a given production order will be used to (partially or totally) satisfy a demand.

A production order can be used to produce intermediate materials or finished materials (that is, finished manufactured products). A production order that makes finished goods may partially or wholly satisfy a single demand, or may satisfy several demands. A production order producing only intermediates does not (directly) satisfy a PPO demand object.

Production orders and material flow arcs are typically part of the optimized solution provided by PPO, but there are times when you provide these objects as part of the data model to be solved. *Firm or fixed information* discusses the situation when part of your production plan is already in effect, cannot be changed, and any newly generated production plan must take that into account.

Production orders may consume or produce a proportional (variable) quantity of material based on batch size, or a fixed quantity of material, independent of batch size. Also, there is great flexibility when specifying material quantities associated with a production order. For example the batch size, material production, and material consumption can all be specified in different units, and you can change these fields using the GUI **Inspector** tool and PPO automatically adjusts the batch size as appropriate.

The planning module supports a constraint on the maximum number of production orders per time interval; this constraint is called the campaign cycle constraint. This is available on the recipe data object. Note also that decisions of resource assignment taken by the planning engine are conserved as part of the production orders, and can be enforced on the scheduling engine (see *Parameters for optimization*).

See also:

♦ Tables in the Production Orders section of the data schema: `PPO_PRODUCTION_ORDER`, `PPO_PRODUCTION_ORDER_ACTIVITY`, `PPO_PRODUCTION_ORDER_PLANNED_MODE`, and `PPO_ACTIVITY`.

- Tables in the Material Flows section of the data schema: `PPO_PROD_TO_DEMAND_ARC`.
- Classes `IloMSAbstractProduction`, `IloMSProductionOrder` and `IloMSAbstractMaterialFlowArc` in the API documentation.
- *Firm or fixed information*.
- In the PPO GUI you can edit production order data using the **Inspector** tool, and data on the **Production Orders** and **Arcs** tabs of the **Transactional Data** view may be available for editing.

Procurements

Procurements are used to represent materials in initial stock or materials procured from outside the plant.

Each procurement corresponds to a given quantity of material with an age according to its optional production date; there is no need to specify a production time if there is no shelf life or maturity constraint. A procurement may override the default shelf life and maturity characteristics of the material. You can also specify a required storage unit to receive the material (optional).

Procurements are the only way to represent initial stock in PPO. Any procurement with a reception date before the start of the model is considered a stock element. The associated production date is used to compute the age of the stock.

See also:

- `PPO_PROCUREMENT` table in the data schema.
- Class `IloMSProcurement` in the API documentation.
- In the PPO GUI on the **Transactional Data** view, procurement information appears on the **Procurements** tab, and may be available for editing.

Firm or fixed information

In practice, there are times during the production planning process when some of the production orders, pegging arcs, and scheduled activities become “firm.” This happens when, for example, production orders have already been released at the plant floor level; produced materials have been assigned to satisfy demands; or scheduled activities have been assigned to specific resources with start and end times. To accommodate this, production orders, material flow arcs, and scheduled activities can provide “firm” information that is respected by the planning, batching and scheduling modules.

A firm production order or arc means that the order (or arc) cannot be abandoned and must be present in the next optimization. Additionally, constraints can be placed on the batch size or quantity involved to ensure that at least a certain amount or no more than a certain amount is provided. Note that although the planning module will not move a firm production order, the scheduling module is able to do so. To prevent the scheduling engine from moving the activities of the production order it must be *fixed*.

Note that a firm production order must be scheduled in order to be taken into account in the planning module (each of its activities must be scheduled).

Note that moving the resource start min forward on the **Gantt Diagram** fixes the production orders scheduled to start before this time fence.

See also:

♦ `PPO_PRODUCTION_ORDER` and any _ARC table in the data schema.

♦ Class `IloMSProductionOrder` and any material flow arc class in the API documentation.

♦ *Production orders and material flow arcs*.

♦ In the PPO GUI on the **Transactional Data** view, you can check for firmed information on the **Production Orders**, **Arcs**, **Planned Productions**, and **Planned Deliveries** tabs.

Extended use of the data model

This section serves as a continuation of *The basic data model*, describing additional modeling considerations and possibilities of Plant PowerOps. The modeling extensions presented here can improve the accuracy and precision of your model. However, some of these features may make the problems more complex to solve, so it is recommended to use only those that significantly improve the plan.

In this section

Production planning considerations
Discusses aspects of modeling data for production planning, including defining your own time buckets and recipe usage.

Resources: Extended usage
Describes additional modeling capabilities when using resources in PPO, including super resources, setups, and cleanups.

Calendars
Calendars are a way to model work shifts, breaks, and other events that affect resource utilization.

Activity constraints, validity periods, and names
Discusses activity constraints, validity periods, and controls over generated activity names.

Material lifespan and inventory
Describes material maturity periods, shelf life, and inventory.

Production planning considerations

Discusses aspects of modeling data for production planning, including defining your own time buckets and recipe usage.

In this section

Inference of the planning problem from the scheduling problem
Only one source of the data model is necessary despite different technologies at work in PPO.

Defining time buckets
You may determine the specific time buckets to best model your problem.

Recipe validity periods
You can specify time periods during which certain recipes can or cannot be used.

Formula optimization and blending
Recipes with flexible ratios of ingredients are allowed.

Inference of the planning problem from the scheduling problem

In PPO there is a single description of the model, and that is the detailed scheduling problem. Although the planning engine uses a very different technology as compared to the scheduling engine (mathematical programming versus constraint programming), there is no need to input the data model twice. For example, a specific mechanism infers the possible routing from the multi-mode recipes.

When considering the planning problem, be aware of:

- The size of time buckets (see *Defining time buckets*) .
- Organizing multiple resources into one super resource (see *Planning with super resources*) .
- Defining a setup approximation level (see *Setup times and costs*).

Defining time buckets

The scheduling module creates a schedule based on enforcing constraints at the level of the declared time unit for the model. The planning engine, however, enforces constraints at time bucket granularity in order to make a longer term production plan.

Defining time buckets is an important task. For example, resource capacity is seen as an energy in the time bucket; inventory max is enforced at each end of the bucket; material balance is global to the bucket; maturity and shelf life constraints are enforced pessimistically with respect to bucket size, and so forth.

By default, the planning module attempts to plan on time buckets that make sense with respect to the remainder of the model. In other words, if you do not define the time buckets, the planning module will create them based on the model data. However, it is strongly recommended that you impose your own time buckets. For example, you might want to generate plans for coming weeks in a detailed manner (using one-day buckets), and for longer-term planning use aggregated buckets (such as one-week buckets).

You can manually create time buckets, starting with the `PPO_BUCKET` table or the `IloMSBucket` class. An easier method is to make PPO generate the necessary time buckets from a template that you create. To do this using a database, see *Automatic bucket generation*, and for API documentation start with the class `IloMSBucketTemplate`.

Note that if a recipe requires a fixed processing time greater than the bucket size, the inference mechanism will automatically detect it and will enforce the resource capacity constraints on a gliding window of several buckets.

Recipe validity periods

You can specify that a recipe shall not be used before or after a certain date. This often occurs when a new recipe replaces an old one. The old recipe is to be used up to a certain date and the new recipe starting from a certain date, generally with a short overlap.

Note that although this could be represented using calendar breaks (see *Calendars*), it is in general simpler to specify earliest and latest start and end times for the usage of recipes, resources, and modes.

In the data schema, columns are provided for this purpose:

♦ In the `PPO_RECIPE` table: `START_MIN` and `END_MAX`, meaning execution of the recipe shall occur between `START_MIN` and `END_MAX`.

♦ In the `PPO_RESOURCE` table: `START_MIN` and `END_MAX`, meaning usage of the resource shall occur between `START_MIN` and `END_MAX`.

♦ In the `PPO_MODE_PROTO` table: `START_MIN`, `START_MAX`, `END_MIN`, and `END_MAX`, meaning the execution of an activity in this mode shall always start between `START_MIN` and `START_MAX`, and end between `END_MIN` and `END_MAX`.

The API classes `IloMSRecipe`, `IloMSResource`, and `IloMSActivity` have methods providing analogous function.

Note that whenever firm information conflicts with the given bounds, the firm information applies and the bound is discarded. Note also that these bounds are all optional with the minimal values defaulting to `-INF` and the maximal values to `+INF`.

Formula optimization and blending

The production planning engine supports recipes with flexible ratios of ingredients for blending purposes. When using this feature the planning engine may create recipe instances as a side effect on the model. For each flexible recipe, a recipe instance with fixed ratios of ingredients may be created for each time bucket. This features allows formula optimization based on quality of materials. The assumption is that qualities blend linearly. See the file `examples/data/oil/refinery.csv` for a blending example.

Resources: Extended usage

Describes additional modeling capabilities when using resources in PPO, including super resources, setups, and cleanups.

In this section

Planning with super resources
The planning module can use aggregated resources to simplify the problem.

Setup times and costs
Resources will often need a setup activity to prepare the resource for use.

Cleanups
Describes the methods available to model the cleaning of resources.

Resource validity periods
You can specify the time periods that resources can be used.

Planning with super resources

A *super resource* is an aggregated group of similar resources with the same connectivity. The reason to create a super resource is to lighten the planning problem, or perform a problem decomposition. Note that only the planning engine uses super resources; the scheduling engine utilizes the individual resource entities as specified in the modes of the activities.

Since the individual resources that compose an aggregated resource are not distinguished by the planning engine, the planning solution may possibly overload an individual resource and under-load another. In practice, if most of the activities that can be executed on a resource *R1* can also be executed on *R2* (and conversely), it is worth creating a super resource consisting of *R1*+ *R2*. If more than half of the production that can go to *R1* cannot go to *R2* (or conversely), then resources *R1* and *R2* should not be grouped.

You can define a super resource as the primary resource of a mode; when the production orders are instantiated from the recipe, PPO automatically generates the alternative modes on the subresources belonging to the super resource. This has multiple benefits:

♦ No need to add alternative modes in a recipe to handle extra resources;

♦ Referring to super resources in modes makes adding new subresources much easier;

♦ In the GUI **Plant Layout** view you can graphically manipulate super resources;

♦ This will factorize the connectivity at the super resource level.

Super-resources are very useful to decompose a problem, and use them wisely in order to split the problem complexity between planning and scheduling. Imagine you are considering how to best model 200 alternative resources. Here are four possible options:

♦ Create one super resource containing 200 resources. Planning will be very easy as there is only one resource to manage, but scheduling has 200 alternatives to consider which is too complex.

♦ Do not use super resources at all; then both planning and scheduling are too complex with 200 alternatives each.

♦ Create 20 super resources with 10 resources each. Planning has 20 resource groups to manage; this seems reasonable. However scheduling still has 200 alternatives to consider.

♦ Do the same as before (20 super resources, 10 resources each) but use the *keep modes of planning* parameter in the PPO settings. Planning will work with alternatives of 20 groups which seems reasonable. Scheduling is not allowed to change the group decided by planning, therefore for each activity you have an alternative of 10 resources which seems reasonable.

Note that since setups in planning are never considered sequence dependent, then planning can have setups on discrete resources as well as unary resources. When you model a super resource including *N* unary resources, the planning engine considers a discrete resource of capacity *N*. If the *N* resources have setups, and you define a setup matrix on the super resources, then the planning engine considers a discrete resource of capacity *N* with setups.

See also:

♦ The `PPO_RESOURCE_HIERARCHY`, `PPO_SETTING`, and `PPO_RESOURCE` tables in the data schema.

- The class `IloMSResource` and `IloMSOptimizationProfile::putSetting` in the API documentation.

Setup times and costs

PPO provides a very flexible framework for representing setup times and costs. Be aware, however, that modeling setups greatly complicates planning and scheduling problems so it is recommended that you represent setup times and costs only when they are significant. To avoid generating an overly complex planning model, it is possible to specify the level of precision at which the planning model must consider setups.

Four setup approximation models are considered:

♦ *Per bucket per recipe* means that the fixed capacity requirements including setups will be counted independently for each bucket and each recipe.

♦ *Cross bucket per recipe* is similar to *per bucket per recipe* except that the continuation of the same recipe from a bucket to the next will not necessitate redoing the setups in the second bucket.

♦ *Per bucket per setup* means that the fixed capacity requirements including setups will be counted independently for each bucket and each setup feature.

♦ *Cross bucket per setup* is similar to *per bucket per setup* except that the continuation of the same setup features from a bucket to the next will not necessitate redoing the setups in the second bucket.

Another option is to ignore setups in production planning.

You can select a different setup model for each resource and, on each resource, even vary the setup model over time. Use the `PPO_RESOURCE_SETUP_MODEL` table or the method `IloMSResource::setPlanningSetupModel` to define the setup approximation.

All these models are approximations of what really happens in the factory. A more precise model would require representation of the sequence of activities that occurs on each resource, that is, to go from a planning to a scheduling model.

Some other considerations when using setups:

♦ Setup times and costs are not assumed to satisfy any triangular inequality; that is, the setup between products A and C might last longer or cost more than the setup from A to B plus the setup from B to C. Nevertheless, whenever triangular inequalities are satisfied, Plant PowerOps exploits them to plan and schedule more efficiently.

♦ Setup times are not required to be smaller than bucket durations. However if a setup time is greater than the time bucket, then the planning engine will use a group of buckets to implement the associated resource capacity constraint.

♦ It sometimes occurs that setups require secondary resources or are subject to calendars that differ from the calendars of production activities. In such cases, it is possible to explicitly add setup activities to the production recipes. Modes are then associated with setup activities exactly as with production activities with two implicit constraints. The first constraint is that in a valid schedule, the setup activity that precedes a production activity must use the same primary resource; secondly, the processing time and cost of the setup activity derive from the setup matrices. Secondary resources and calendars can then be associated with each mode of the setup activity. Precedence constraints can also be associated with setup activities if, for example, a minimal or maximal delay applies between the setup activity and the corresponding production activity.

- Setups can be described along several features. A typical case in which this is useful is a liquid product packaging line: (a) the configuration of the line changes with the packaging and (b) the pipes that feed the line have to be cleaned or changed when the product to be packaged changes. Rather than describing the transitions between all of the (packaged product, packaging) combinations, Plant PowerOps allows you to provide times and costs for the change of packaging on the one hand, and times and costs for the change of packaged product on the other. Setup features are additive; so when both the packaging and the packaged product change, the overall setup time is the sum of the two setup times and the overall setup cost is the sum of the two setup costs.

See also:

- The `PPO_ACTIVITY_SETUP_STATE_PROTO`, `PPO_RESOURCE_SETUP_MODEL`, `PPO_SETUP_ACTIVITY_PROTO`, `PPO_SETUP_MODE_PROTO`, `PPO_SETUP_MATRIX`, and `PPO_RESOURCE_SETUP_STATE` tables in the data schema.
- The classes `IloMSSetupMatrix` `IloMSSetupActivity`, and `IloMSResource`, and the enumeration `IloMSCleaningStatus` in the API documentation.
- *Modeling a dairy plant with PPO Java API* for an extended example that includes setups.
- In the PPO GUI, the **Resources** tab on the **Master Data** can be edited to change some setup requirements.

Cleanups

The cleaning of resources is a vital part of manufacturing plans and schedules. Cleaning is modeled in PPO using production orders of a cleanup recipe. A cleanup recipe is composed of an activity with a cleaning status (`PPO_ACTIVITY|CLEANUP_STATUS` or `IloMSCleaningStatus.Cleaning`). The cleaning time is the processing time of this activity. The cleaning cost is the processing cost of the cleaning recipe.

Some cleanup requirements are specified on the resource: Clean the resource every ten hours, or after five hours of idle time, or every ten batches. Some cleaning requirements are based on the product transition or setup state: Clean the resource after processing an allergen, or clean the resource as part of the setup to process a certain product. These latter requirements are specified in a setup matrix.

You can use the `PPO_RESOURCE_CLEANUP_STATE` table, the `IloMSResource` class, or the GUI (**Master Data > Resources > Edit** resource) to specify the following cleanup requirements:

- ♦ Maximum time before a cleanup: This specifies the maximal number of time units that a resource can remain without a cleanup. After this time value, a cleanup is required before any subsequent resource usage. This time is counted from the end time of the previous cleanup. The default is infinity.
- ♦ Maximum idle time before a cleanup: This specifies the maximal number of time units that a resource can remain idle without requiring a cleanup. After this time value, a cleanup is required before any subsequent resource usage. The default is infinity.
- ♦ Maximum number of batches before a cleanup: This specifies the maximal number of batches that can be processed between two cleanups. The default is infinity.

You can use the `PPO_SETUP_MATRIX` table or `IloMSSetupMatrix` class to specify whether a cleanup is required in between the processing of two production activities on the same resource (that is, in the transition from one resource state to another). The default for all transitions is false.

There are some requirements when modeling cleanups in PPO. The cleanup recipe must have one and only one activity, and that activity must have a cleanup status of cleaning. The activity must also have one and only one mode executable on the resource under consideration, which defines the time and cost, and possibly the secondary resources, required to clean the resource.

See also:

- ♦ The `PPO_ACTIVITY`, `PPO_SETUP_MATRIX`, and `PPO_RESOURCE_CLEANUP_STATE` tables in the data schema.
- ♦ The classes `IloMSSetupMatrix` and `IloMSResource`, and the enumeration `IloMSCleaningStatus` in the API documentation.
- ♦ *Modeling a dairy plant with PPO Java API* for an extended example that includes cleanup requirements.
- ♦ In the PPO GUI, the **Resources** tab on the **Master Data** can be edited to change cleanup requirements.

Resource validity periods

There are two main ways to control the temporal availability of resources. You can use calendars, as described in *Calendars*. There is a simpler way that might be more applicable in certain situations. For example, if you're replacing an old machine with a new one it might be easier to just specify the earliest start and latest end times for the resources.

The `PPO_RESOURCE` table has the fields `START_MIN` and `END_MAX`, meaning usage of the resource shall occur between `START_MIN` and `END_MAX`. In the API, the methods `IloMSResource::setStartMin` and `IloMSResource::setEndMax` provide that function.

Note that whenever firm information conflicts with the given bounds, the firm information applies and the bound is discarded.

Calendars

Calendars are a way to model work shifts, breaks, and other events that affect resource utilization.

In this section

Why use calendars?
Calendars are vital for modeling resource availability.

Calendar intervals
Each calendar is composed of calendar intervals.

Overlapping calendar intervals
Combined interval capacities can either be added or constrained.

Calendars on modes or resources
Calendars can be associated with modes and with resources.

Breaks and shifts
Calendar intervals can be breaks and end of shift periods.

Productivity
The productivity directly affects activity duration.

Why use calendars?

In most settings, resources do not operate homogeneously over time:

- The capacity of a resource may vary over time; for example, in a given work center five machines may operate during the day but only two during the night.
- The productivity of a resource may vary over time; for example, the human operator is slower at night.
- Production may be stopped during lunch periods, breaks, or shift changes.
- In some cases, it is not possible to start an activity in one shift and finish it in another, for example, because one must track during what shift the production was performed.

Plant PowerOps allows you to define calendars to take all these issues into account.

Calendars can be associated with modes, or assigned directly to resources. There are different effects depending which is used, and this is discussed in *Calendars on modes or resources*. By default, calendars are assigned neither to modes nor to resources.

Calendar intervals

Each calendar is composed of calendar intervals, and each interval has properties of resource capacity, productivity, break times, and end of shift characteristics. The default values for these properties are that the default capacity is the resource capacity to which the calendar is attached; the default productivity is 1.0; and by default, a calendar interval is neither a break nor an end of shift.

Calendar intervals are often intended to be used repeatedly: the day shift, the night shift, the maintenance schedule, and so on. If you create calendar intervals with a *periodicity*, then you create one model interval that is repeated according to your specifications. This lets you easily set the desired characteristics for an entire shift, and as necessary, easily modify the shift properties because the periodic intervals remain linked as a single modeling object.

If the interval has a periodicity of zero, then it appears only once in a calendar; but it is possible to copy this *aperiodic* interval in the PPO GUI **Calendars** view. The copies of the original aperiodic interval are not linked; so if you change one, then you change only that one interval, not its unlinked copies.

Overlapping calendar intervals

When calendar intervals overlap in time, there are two possibilities for calculating the resulting resource capacity. Either the capacities can be added together, or the most constraining interval dictates the available capacity. If the calendar is *additive*, and several calendar intervals overlap, then the corresponding capacities from each interval are added together to determine the actual capacity of the resource at any given time. If the calendar is not additive, then the most constraining capacity from the overlapping intervals determines the (limited) capacity available at any given time.

To prevent confusion, it is recommended to avoid using overlapping intervals if the calendar can easily be represented in such a manner. Note that in all cases, limitations induced by the capacity provided by the resource itself (as modeled with `PPO_RESOURCE`, `IloMSResource`, and so forth) also apply. The *additive* property applies only to resource capacity, not to productivity.

Calendars on modes or resources

Calendars can be associated with modes and with resources. If a mode has a calendar, then this calendar applies to activities executed in this mode. If the mode does not have a calendar but the primary resource of the mode does have a calendar, then the calendar of the resource applies to the activities executed in this mode. You can think of the calendar on the resource as the "default" calendar used when no mode calendar has been specified.

Calendars provide slightly different modeling capabilities depending on whether they are attached to the mode or the resource. Breaks and work periods are taken into account on calendars assigned to modes, but resource capacities are not. Resource capacities are taken into account by calendars on resources, and for determining capacity limitations, the calendars associated with resources always apply.

Note that for “block planning,” it is useful to use calendars on modes.

As mentioned, when the calendar is associated with the mode, then breaks are taken into account. See *Breaks and shifts* for more information.

Breaks and shifts

You can specify that a calendar interval is a break. Breaks are time intervals when activities generally cannot be executed because, for example, a resource is not available. So you could use a calendar interval to model a lunch break, or a daily afternoon worker break of 15 minutes.

You can specify that a calendar interval is an end of shift period; for example, an interval ending at 5:00 p.m. is the end of the day shift. Some activities cannot be executed across a shift change. If a calendar has no shift intervals, it is assumed that there are no shifts (or equivalently that all activities executed in the calendar are breakable by shifts).

When the calendar is assigned to a mode, there are additional controls that you can provide in regards to breaks and shifts: maximal break duration, maximal end duration in a break, and shift-breakable permission.

The *maximal break duration* (`PPO_MODE|BREAK_DURATION_MAX`) specifies the longest duration that a break can be and not interrupt an activity which is executing in this mode. If a break lasts longer than the maximal break duration, then the activity must either be completed before the break, or it must start after the break. If a break is less than the maximal break duration, then the activity can be interrupted by the break, with no corresponding restrictions on the activity start or end time. If the break maximal duration is zero, then the activity is not breakable.

The *maximal end duration in a break* setting is used to specify that it is permissible to continue an activity into a break period if the activity can be completed within a certain time limit after the start of the break. This can be set by the `PPO_MODE|MAX_END_DURATION_IN_BREAK` field.

An activity that is *shift-breakable* can overlap a shift change; an activity that is not shift-breakable must be completely executed within a single shift. So if a calendar interval is an end of shift, then a non-shift-breakable activity cannot both start before and end after the end time of this interval.

Productivity

The productivity of an interval specifies the speed at which the activity executes over time. By default, productivity is equal to 1.0. The productivity is used to relate the processing time and the duration of the activity. When the productivity is 0.5, two units of duration are necessary to execute one unit of processing time. When the productivity is 2.0, one duration unit will execute two units of processing time.

When the productivity is greater than 1.0, the processing time of the mode will usually not be fixed; that is, the difference between the maximal processing time and the minimal processing time will usually exceed "productivity - 1.0" in order to allow an appropriate rounding of both the processing time and the activity duration. For example, if in a solution the productivity is 2.0 from the start time to the end time of the activity, and if the duration is 10, then the processing time in this solution is 20. If the minimal and maximal processing times are both equal to 19, then the engine would regard this solution as invalid.

See also:

- `PPO_CALENDAR`, `PPO_CALENDAR_INTERVAL`, `PPO_MODE`, and `PPO_RESOURCE` tables in the data schema.
- Classes `IloMSCalendar`, `IloMSCalendarInterval`, `IloMSMode`, and `IloMSResource` in the API documentation.
- An C++ example of modeling a problem that uses calendars *Using the PPO API for C++ to model and solve*.
- A Java™ example of modeling a problem that uses calendars at *Using the PPO API for Java to model and solve*.
- In the PPO GUI, the **Calendars** view lets you visualize and edit calendars information; right-click to display the contextual menu. Also on the **Master Data** view, **Resources** tab, you can select the calendar a resource uses.

Activity constraints, validity periods, and names

Discusses activity constraints, validity periods, and controls over generated activity names.

In this section

Precedence constraints
Describes precedence constraints between activities.

Spanning constraints
Describes spanning constraints on activities.

Compatibility constraints
Describes constraints that enable you to impose that two activities be performed in related ways.

Activity validity periods
You can specify the time period during which certain activities can be performed.

Activity names
You can control the names of the generated activities.

Precedence constraints

You can define precedence constraints between two activities of the same recipe, referred to as the “predecessor” activity and the “successor” activity. The activities can be production or setup activities. Four types of precedence constraints are distinguished:

- ♦ EndToStart is in many cases the only type of precedence constraint that is needed. An EndToStart constraint means that the predecessor activity must end before the successor activity starts or, equivalently, that the successor cannot start before the end of the predecessor. A minimal delay and a maximal delay between the end of the predecessor and the start of the successor can also be imposed.
- ♦ StartToStart means that the predecessor activity must start before the successor activity starts or, equivalently, that the successor cannot start before the start of the predecessor. A minimal delay and a maximal delay between the start of the predecessor and the start of the successor can also be imposed.
- ♦ StartToEnd means that the predecessor activity must start before the successor activity ends or, equivalently, that the successor cannot end before the start of the predecessor. A minimal delay and a maximal delay between the start of the predecessor and the end of the successor can also be imposed.
- ♦ EndToEnd means that the predecessor activity must end before the successor activity ends or, equivalently, that the successor cannot end before the end of the predecessor. A minimal delay and a maximal delay between the end of the predecessor and the end of the successor can also be imposed.

See also:

- ♦ `PPO_PROD_PROD_PRECED_PROTO` and other `*_PRECED_PROTO` tables in the data schema.
- ♦ Classes `IloMSPrecedence` and `IloMSAbstractActivity` in the API documentation.

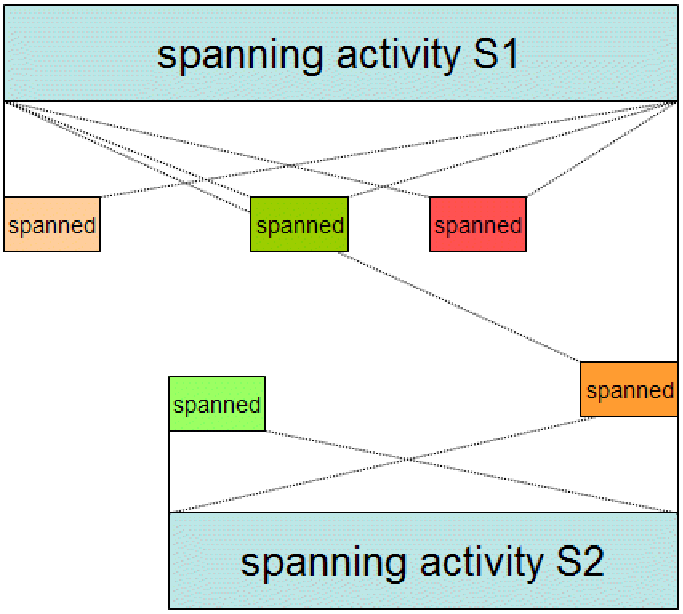
Spanning constraints

The *spanning constraint* can be considered to be a special type of precedence constraint. This constraint spans over multiple activities such that one activity spans over the elapsed times of others. The time bounds of the spanning activity are computed as a propagation of the time bounds of its spanned activities. More precisely, it means that the start time of the spanning activity is equal to the earliest of the start times of the spanned activities, and that the end time of the spanning activity is equal to the latest of the end times of the spanned activities.

The main difference between precedence and spanning constraints is that with precedence constraints, although one can specify that an activity “covers” the elapsed times of a set of activities, if there is no known activity order then one cannot constrain the covering activity so that:

- The start time of the covering activity is equal to the start of the earliest covered activity;
- The end time of the covering activity is equal to the end time of the last covered activity.

The spanning constraint may be enforced only on activity start, only on activity end, or on both.



See also:

- `PPO_SPANNING_PROTO` table in the data schema.
- The class `IloMSActivity` in the API documentation.

Compatibility constraints

Compatibility constraints refers to a collection of restrictions that you can enforce on modes and activities. The activities can be production or setup activities.

You can constrain:

- That the two modes (of the two activities) have the same primary resource;
- That the two modes have the same primary resource and the same capacity requirement;
- That the two modes have the same line identifier;
- That the primary resources of the two activities are connected;
- That the two activities must have the same (or opposite) performed status;
- That the performed status of one activity logically implies the status of the second.

When one states that two activities must execute on same line identifier, then the optimizer will choose modes that share a common line id.

Resource connections

The *resource connection* is used to define physical connections between resources so that one can enforce the usage of two connected resources as primary resources of two activities.

When one states that two activities must execute on connected primary resources, then the optimizer will enforce that their respective modes refer to primary resources for which a connection has been defined. To specify the connections use the `PPO_RESOURCE_CONNECTION` schema table or the `IloMSResource::addConnectedResource` method.

Note that a resource **A** is connected with a resource **B** if one of the following is true:

- **A** is directly connected with **B**
- **A** is directly connected with superResource(**B**)
- superResource(**A**) is directly connected with **B**
- superResource(**A**) is directly connected with superResource(**B**).

See also:

- `PPO_PROD_PROD_COMPAT_PROTO`, `PPO_RESOURCE_CONNECTION`, `PPO_MODE_PROTO`, `PPO_ACTIVITY`, `PPO_RESOURCE` and `*_COMPAT_PROTO` tables in the data schema.
- Classes `IloMSActivityCompatibilityConstraint`, `IloMSActivity`, `IloMSMode`, `IloMSResource`, and enumeration `IloMSActivityCompatibilityType` in the API documentation.

Activity validity periods

You can specify that an activity mode cannot be used before or after a certain date. This could be useful, for example, if one activity mode is obsolete and a new one is replacing it.

You can control this through calendars (see *Calendars*) but there is an easier method in the relational and object models. In the `PPO_MODE` table, you can use the fields `START_MIN`, `START_MAX`, `END_MIN`, and `END_MAX`, which mean that the execution of an activity in this mode shall always start between `START_MIN` and `START_MAX`, and end between `END_MIN` and `END_MAX`. In the API, the class `IloMSActivity` has methods for analogous function.

Also of potential interest, from the relational model:

- In the `PPO_RECIPE` table: `START_MIN` and `END_MAX`, meaning execution of the recipe shall occur between `START_MIN` and `END_MAX`.
- In the `PPO_RESOURCE` table: `START_MIN` and `END_MAX`, meaning usage of the resource shall occur between `START_MIN` and `END_MAX`.

Note that whenever firm information conflicts with the given bounds, the firm information applies and the bound is discarded. Note also that these bounds are all optional with the minimal values defaulting to `-INF` and the maximal values to `+INF`.

Activity names

The name you give in the model to an activity prototype (for example on `PPO_ACTIVITY_PROTO|NAME`) is not, by default, the name that will appear on the **Gantt Diagram** for the scheduled activities cloned from this prototype. Rather, activities generated from activity prototypes have an automatic long name based on a concatenation of the production order name and the activity prototype name, separated by a dot.

To override the default and use the original name of the activity prototype that you modeled, you must set the setting `bShortName` to true. When the `bShortName` setting is true, the activity label from the activity prototypes interprets the following special characters as shown:

- ^a = abbreviated main product name to 3 first letters (does not support multibyte characters)
- ^b = abbreviated main ingredient name to 3 first letters (does not support multibyte characters)
- ^i = main product identifier
- ^j = main ingredient identifier
- ^m = main ingredient name
- ^n = order index
- ^o = order identifier
- ^p = main product name
- ^q = quantity of main product or batch size; if missing round up as an integer
- ^r = recipe name, or if missing then the recipe identifier, or if missing the recipe id
- ^s = batch size

You can define multiline names using `\n` to start a new line. To properly see multiline names in the **Gantt Diagram**, increase the row height using the **Activity and Resource Chart Row Height** in the **Tool > Options** menu in the GUI.

See also:

- `PPO_ACTIVITY_PROTO` and `PPO_SETTING` tables in the data schema.
- The class `IloMSActivityCalendar` and method `IloMSOptimizationProfile::putSetting` in the API documentation.

Material lifespan and inventory

Describes material maturity periods, shelf life, and inventory.

In this section

Maturity, shelf life, waste recipes
Discusses maturity, shelf life, and the use of waste recipes to remove obsolete inventory.

Managing inventory stock
Managing stock levels is a vital part of a manufacturing plan.

Maturity, shelf life, waste recipes

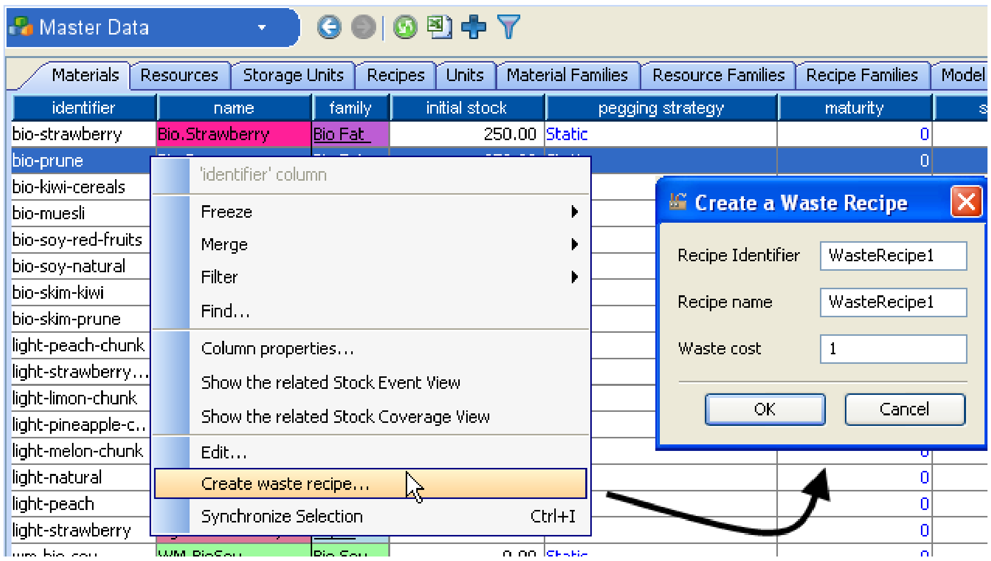
The planning and scheduling engines of PPO allow you to set the time window during which a material is consumable. Maturation is defined as the minimal number of time units that must elapse after material production completes before that material can be consumed. Shelf life is defined as the number of time units before the material expires and is no longer consumable. The default maturation time is zero and the default shelf life is infinite. Production orders and demands can consume only mature, non-expired items. Demands can also specify a requirement that materials have a minimal remaining shelf life.

The age of the item is computed starting at the end of the producing activity of a production order, or at the specified production time of a procurement. Maturity and shelf life constraints are handled pessimistically by production planning. For instance if a material shelf life is less than the size of a bucket, it must be consumed in the same bucket as it is produced. Note that in the case of a procurement, this age is dependent upon the receipt time at which the material enters inventory. One can override the maturity and shelf life of a procured item using the `PPO_PROCUREMENT` table or `IloMSProcurement` class.

Waste recipes

By default, obsolete material is held in inventory forever. In order to discard expired material, one must define a *waste recipe*. A waste recipe is simply a recipe that consumes the material and produces nothing. The waste cost is the cost incurred to throw away one unit of material. The waste recipe is typically composed of one activity prototype consuming the material with a minimum batch size of zero and an infinite maximum batch size. This activity prototype may have several modes, each being a way of consuming the material from the different storage units. Note that when using the API there is a helper function to help you create a waste recipe in one call; see the method `newWasteRecipe` in the `IloMSModel` class.

The contextual menu on the **Materials** tab of the **Master Data** view allows you to create a waste recipe directly in the GUI.



identifier	name	family	initial stock	pegging strategy	maturity
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	0
bio-prune					0
bio-kiwi-cereals					
bio-muesli					
bio-soy-red-fruits					
bio-soy-natural					
bio-skim-kiwi					
bio-skim-prune					
light-peach-chunk					
light-strawberry...					
light-limon-chunk					
light-pineapple-c...					
light-melon-chunk					
light-natural					0
light-peach					0
light-strawberry					0
wm-bio-sou	WM-BioSou	Bio Sou	0.00	Static	0

Recipe Identifier	WasteRecipe1
Recipe name	WasteRecipe1
Waste cost	1

See also:

- The `PPO_MATERIAL` and `PPO_PROCUREMENT` tables in the data schema.
- The classes `IloMSMaterial` and `IloMSProcurement` in the API documentation.
- *Materials*
- *Managing inventory stock*
- An example of the `PPO_MATERIAL` table in *Step 5 Define the materials* of *Modeling a simple problem: A “bottleneck” resource*.

Managing inventory stock

Managing stock levels is a vital part of a manufacturing plan.

In this section

Overview of three approaches
Planners must balance the costs of maintaining inventory with having enough product to satisfy customer demand.

Material inventory costs
Modeling numerous levels of inventory, each with a particular cost.

Safety inventories
Discusses the method of maintaining a minimal inventory.

Setting days of supply targets
Typically a range of acceptable inventory levels and costs is specified.

Service levels
How service levels provide superior management of inventory.

Overview of three approaches

Planners must balance the costs of maintaining inventory with having enough product to satisfy customer demand. The ultimate goal is too ensure that inventory levels are just right; not too big, thereby causing unnecessary expense, and not too small, thereby risking stock outages.

PPO provides three increasingly sophisticated methods of achieving those goals. The first method applies cost functions directly to inventory levels. You can create threshold levels in the cost function; if, for example, inventory at a certain size requires you to access a new storage facility, it's easy to model that cost increase. Conversely, if inventory gets so small that you risk a stock outage, you can model an increased potential cost (risk of unsatisfied customers) at the lower threshold. This method is described in *Material inventory costs* and *Safety inventories*.

Another method is to apply a cost function based on demand coverage, expressed in days of supply (DOS). The point is to create a target inventory corridor that is allowed to change over time; the change can be due to different DOS targets or to demand fluctuation. Production may have to be tuned in order to stay within the corridor. This method is described in *Setting days of supply targets*.

These methods can be used to manage inventory levels, however there are some shortcomings. You have to predict and compute the customer demand levels in advance yourself, and tune production as necessary. This can lead to forecast error. Neither technique reliably accounts for variability in production lead time. Neither lets you directly manage stock using industry-standard service levels, which allow you to predict or control the potential of running out of stock.

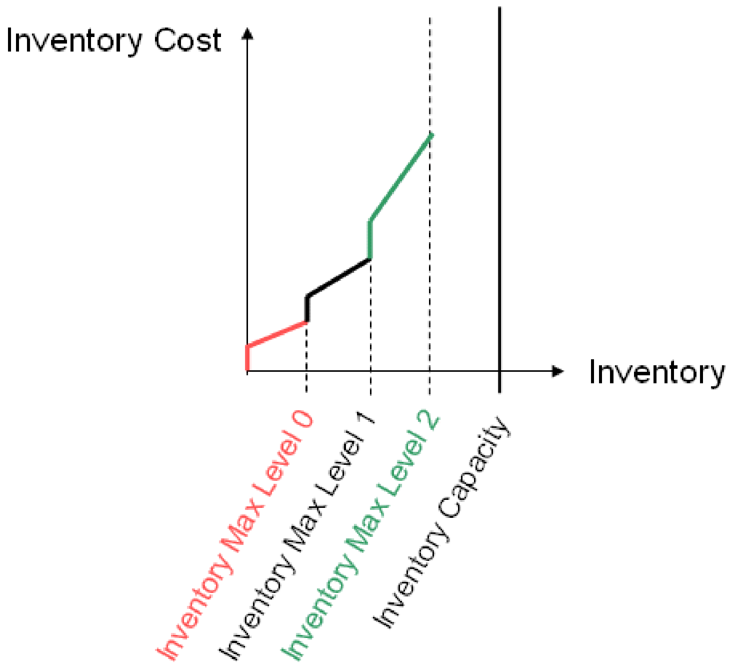
The section *Service levels* introduces the PPO technology available to handle those concerns.

Material inventory costs

For each instance of material inventory, one or several cost functions with different periods of validity can be provided. This allows the cost of maintaining inventory to depend on time, such as due to seasonal effects. Note that the same functions may be used for different materials.

In most cases, the cost of inventory varies linearly with the amount of material in inventory. In some cases, however, some threshold effects occur, such as when the inventory goes over a given limit, an outside tank must be rented thereby increasing the cost. For this reason, inventory cost functions are represented in levels; each level is described by a maximal inventory for this level, a fixed cost for entering this level, and a variable cost paid for each unit of material throughout this level. Needless to say, optimization is easier when there is a unique level and no fixed cost, so it is recommended to use multiple levels only when cost variations are significant.

Both the fixed and the variable costs are paid per unit of time. The following image represents the fixed and variable costs for each level of inventory.

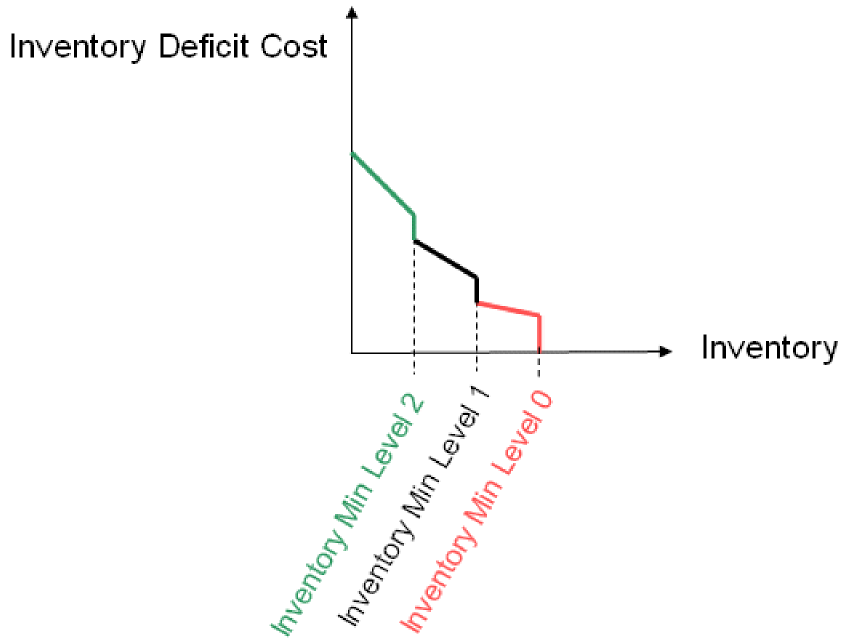


In the relational model, see the tables `PPO_MATERIAL`, `PPO_INVENTORY_MAX_COST_FCT`, and `PPO_INVENTORY_MAX_COST`. In the API documentation see the classes `IloMSInventoryMaxCostFunction` and `IloMSMaterial`.

Safety inventories

Preferences for maintaining safety stock (or minimal) inventories can be represented in a similar manner to that used for maximal inventory costs.

The following image represents the fixed and variable costs for each level of minimal or safety stock inventory.



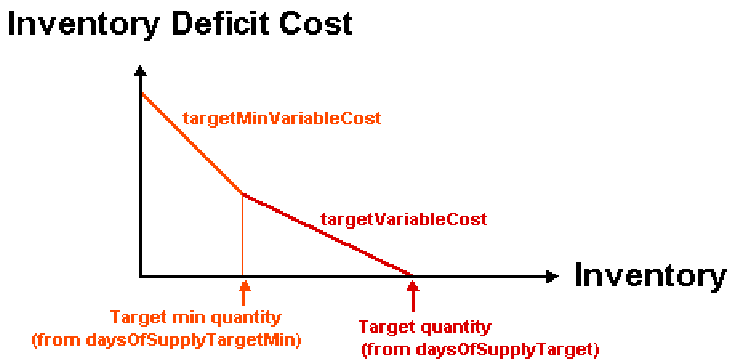
As for maximal inventory costs, both the fixed and the variable costs are paid per unit of time. In the relational model, see the tables `PPO_MATERIAL`, `PPO_INVENTORY_MIN_COST_FCT`, and `PPO_INVENTORY_MIN_COST`. In the API documentation see the classes `IloMSInventoryMinCostFunction` and `IloMSMaterial`.

Setting days of supply targets

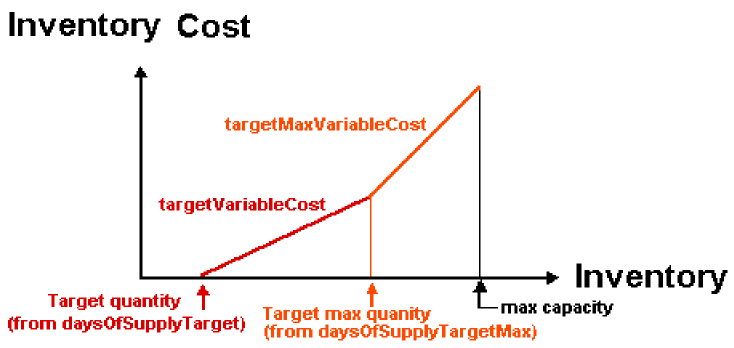
For some facilities, it is convenient to set inventory cost functions based on demand coverage, expressed in days of supply (DOS). PPO provides a shortcut for that case. You provide the minimum and maximum days of supply targets, along with the corresponding variable costs per unit of material for exceeding those DOS bounds. A maximal inventory for the material can also be provided. For short term planning, this data is sufficient for PPO to automatically derive the inventory cost functions.

For midterm planning, an additional level in-between the min and max can be defined. This level is called the *days of supply target*, and represents the ideal number of days of supply that you want to hold in inventory at the end of each time bucket. This value must be between the min and max, is considered a strong preference but not necessarily obeyed, and is best used when the time bucket duration is greater than the width of the stock corridor (between min and max).

Using the notion of *target minimum days of supply* to define the inventory min cost creates a cost function like the following.



Using the notion of *target maximum days of supply* to define the inventory cost creates a cost function like the following.



See the table `PPO_MATERIAL` in the data schema or the class `IloMSMaterial` in the API documentation for more information.

Service levels

In PPO you can control stock levels by your desired service level and service level type. This can include calculation of lead time, safety stock, and variability in demand and production time.

For each controlled material provide:

- A *service level target* which is a percentage between 50% and 100%. 50% represents no safety stock;
- A *service level type* to interpret the semantics of the target, based on industry-standards of *cycle service level* (alpha, type 1, or event-based level) and *fill rate* (beta, type 2, or quantity-based level);
- A demand variability ratio to express demand uncertainty including forecast error;
- An average lead time with a standard deviation.

The alpha and beta service levels also have *dynamic* versions, which are based on the time bucket of the next production, not the average lead time. The PPO planning module supports dynamic safety stock where the safety stock is a decision variable depending on the decision regarding the next production.

See *Using service levels, lead time, and demand variability to manage stock levels* for more information.

By default, the service level type is disabled, and the method of entering days of supply targets is used to define a stock corridor (see *Setting days of supply targets*).

Applied use of Plant PowerOps

This section describes various aspects of using and applying IBM® ILOG® Plant PowerOps (PPO), focusing on tasks involving the GUI. The first section introduces general basic tasks in the GUI. This is followed by topics describing database usage, using service levels to manage inventory, problem decomposition, and editing production plans.

In this section

Using the PPO GUI
Description of some tools, tasks, and views of the PPO GUI.

Database usage and connectivity
This describes the supported databases of PPO and their usage; for advanced topics see *Database customization*.

Using service levels, lead time, and demand variability to manage stock levels
This section describes how to manage stock inventory levels to help prevent stock outages even when faced with uncertain demand and production time.

Decomposition framework
How to decompose a planning problem into smaller components.

Production planning simulations
The planning solution is editable in the PPO GUI and allows you to run simulations and re-optimize with planning.

Advanced usage: Distribution planning
This advanced section describes how to use PPO to optimize your distribution network. Any usage of PPO for distribution planning, supply chain planning, or multi-plant planning must be validated with the PPO product management team.

Using the PPO GUI

Description of some tools, tasks, and views of the PPO GUI.

In this section

GUI tool and navigation tips
An overview of some of the tools and aids to navigation in the PPO GUI.

Stock Coverage view
Describes the Stock Coverage view.

The Inspector
Describes some aspects of the Inspector tool.

The Parameters window
Describes the selections on the Parameters window.

Using the checker
Describes how to use the **Checker** to test your plan and model data.

Copying an existing recipe
As recipes can become complex, it's sometimes easier to copy a recipe and modify it as necessary, as opposed to creating a new recipe.

GUI tool and navigation tips

An overview of some of the tools and aids to navigation in the PPO GUI.

In this section

Menus and toolbars
This section describes the menus and toolbars in the IBM® ILOG® Plant PowerOps (PPO) graphical user interface.

Accessing the plan views
Using the *view types* bar.

Splitting and synchronizing plan views
Allows visualization of more scenario data.

Using the Filter tool on the Master and Transactional Data tables
You can filter by recipe or resource.

Comparing multiple scenarios
New horizontal or vertical groups.

Repair extent, capacity and magnetism
Describes the use of *repair extent, repair capacity* and *magnetism* in Plant PowerOps.

Menus and toolbars

This section describes the menus and toolbars in the IBM® ILOG® Plant PowerOps (PPO) graphical user interface.

In this section

Overview
Provides an overview of the user interface menus and toolbars.

The menu bar
Describes the menus available on the main menu bar.

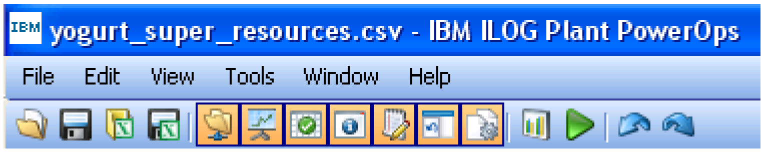
Main toolbar
Describes the row of icon tools at the top of the PPO interface.

Plan View toolbar
Describes the icon tools available on the Plan View toolbar.

Overview

Menus and toolbars provide your control over and interaction with PPO.

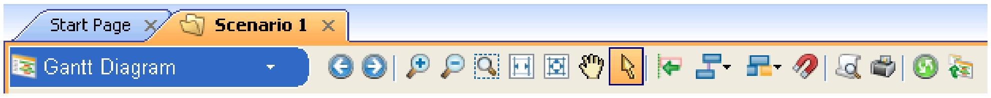
The following image shows the menu bar (**File**, **Edit**, and other menus) and the main toolbar (the row of icons).



After you select an icon on the toolbar, the icon switches from its standby state with a blue background to an orange background.

See *The menu bar* and *Main toolbar* for information about the selections and icons above. The exact menus and icons that are active depend upon which windows you have open. Not all items are available in all views

Although there is only one menu bar in PPO, there are numerous toolbars. Shown here is the toolbar for the **Gantt Diagram** view.



See the *Plan View toolbar* for information about these tools.

The menu bar

The menu bar has the following menus:

The File menu

The Edit menu

The View menu

The Tools menu

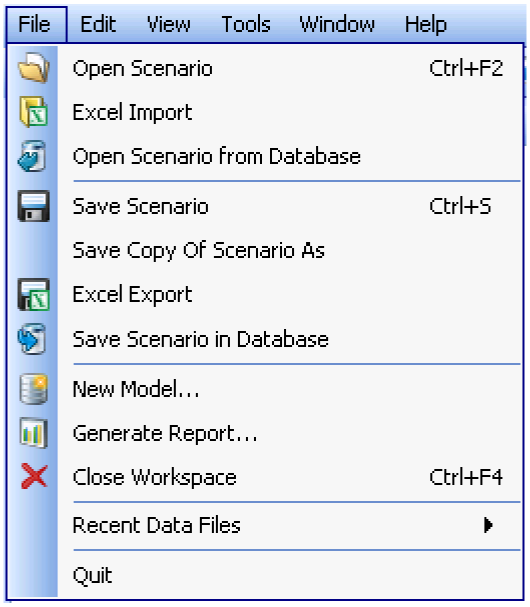
The Window menu

The Help menu

The File menu

The first selections on this menu allow you to bring data into the GUI. The **Open Scenario** selection opens a file browser so that you can locate and open an existing problem data file in .mdb or .csv format. The **Excel Import** selection allows you to import Microsoft® Excel® spreadsheet files. Use **Open Scenario from Database** to input plan data from an Oracle® or Microsoft database. The next group of selections provide methods to save or export your data from PPO in different formats. The **Excel Export** selection allows you to export data to Excel spreadsheet files.

More information on using PPO with a database is available in the documentation set. Note also that several options regarding database files are set on the **Parameters** page on the **Options** window.



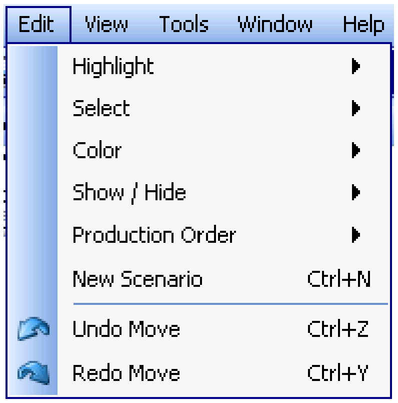
New Model... opens the **New Model Wizard**, which helps you create a new problem model in the GUI itself, bypassing the need to build a database or spreadsheet model. **Generate Report...** is useful to get a comprehensive look at capacity, demand and inventory using Tableau.

Close Workspace will close the currently active workspace and all scenarios contained within, but not close other problems you may have open.

Recent Data Files provides a shortcut to files that you may have worked with recently. **Quit** will close PPO.

The Edit menu

The appearance of the **Edit** menu changes depending on what window in PPO is active. The following image shows all available choices.



The **Highlight** selection lets you reveal production orders, activities, arcs and constraints on the **Gantt Diagram**. Note that this choice displays or blinks items on the Gantt but does not *select* them.

Select does allow you to select items on the view displayed on the **Gantt Diagram**. This includes activities highlighted from the previous **Highlight** menu item, the start min, and more. **Color** allows you to reset the colors displayed in the **Gantt Diagram** in order to better visualize information in the plan.

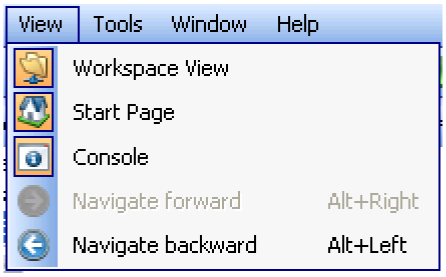
Show/Hide also works with the **Gantt Diagram**; if an item is selected on the Gantt chart, then selecting **Show —> Primary product stock coverage** displays the **Stock Coverage** view for the primary product of that selected activity or production order. **Show Pegging arcs** reveals the material flow arcs for the selected production orders on the Gantt chart. There is also a selection available to hide any displayed arrows and turn off any highlighting.

New Scenario allows you to create a new scenario for the problem; use this choice to have two or more scenarios open in the interface for which you can generate independent plans.

The last two choices on the **Edit** menu are undo and redo. The image above shows **Move** because the last action performed by the user was to move an object on the Gantt chart.

The View menu

The **View** menu allows you to display or hide various windows of the PPO GUI. An orange background to the icon (as shown for the **Start Page** and **Console** below) indicates that the item is currently displayed.

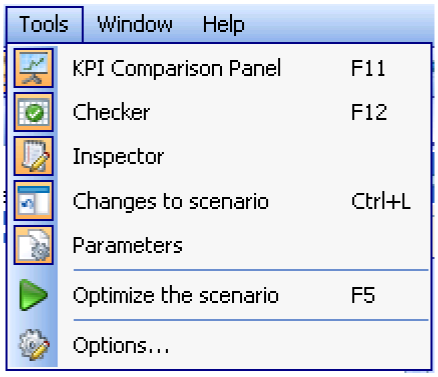


Hiding the **Workspace View** does not close the problem or scenarios, it merely hides that window in order to provide more space in the application window.

The **Console** is the communications window of PPO which lists details of optimization and other information. The **Navigate** choices allow you to page forward and backward through the (already-displayed) views within the *active* scenario.

The Tools menu

The **KPI Comparison Panel** selection displays a table that allows you to compare the values of the Key Performance Indicators across numerous scenario plans. KPIs are discussed in the documentation set.



Checker displays a window that keeps you informed of any error messages; selecting an error message in the Checker window highlights the associated constraints, activities and other items in the **Gantt Diagram**.

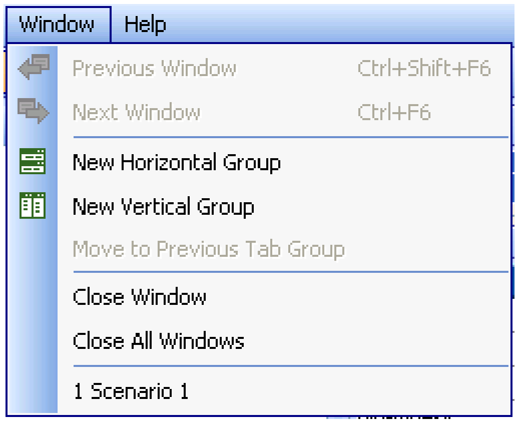
The **Inspector** is an information window used to display details of activities and other items on the Gantt chart; you can edit information colored in blue in this window. See *The Inspector* for more information.

The **Changes to scenario** window keeps track of changes you make to the plan, and provides undo and redo commands. The **Parameters** window lets you organize how inventory and material information is displayed in various views; items colored in blue in the Parameters window provide scroll boxes with available options. See *The Parameters window* for more information.

Optimize the scenario displays a dialog box that allows you to direct PPO to find a solution for production planning, batching, and scheduling. Finally, you can set various **Options** regarding the views, magnetism, database usage, and traces.

The Window menu

The **Window** menu provides controls for you to customize and navigate through the open Plan View windows in the interface.



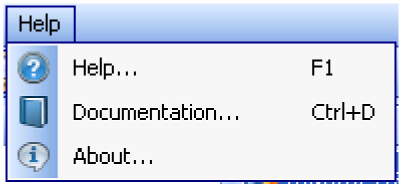
The selections **Previous Window** and **Next Window** allow you to page backwards and forwards through the open scenarios in the active workspace.

The selections **New Horizontal Group** and **New Vertical Group** allow you to arrange the scenario windows into two horizontal or vertical layouts. There must be at least two scenarios open for these commands to function. See *Comparing multiple scenarios* for more information.

Move to Previous Tab Group restores the scenarios to one view. The list of files at the bottom provides a quick link to the various open scenarios.

The Help menu

The **Help...** selection displays the *IBM ILOG Plant PowerOps Online Help*.

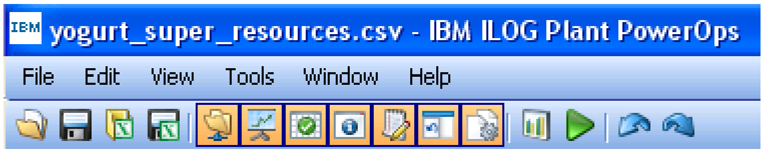


The **Documentation** selection displays the documentation set. The documentation includes implementation information, reference manuals, entity relationship diagrams, the data schema, and more. Documentation in CHM features search across the full documentation set. Individual publications in CHM and PDF format are available in the <*PPOInstallDirectory*>\doc\chm or <*PPOInstallDirectory*>\doc\pdf directories. HTML documentation is available at <*PPOInstallDirectory*>\doc\html\en_US\documentation.html.

The **About...** selection displays information about your PPO installation.

Main toolbar

The main toolbar is the row of icons at the top of the PPO interface. It gives you quick access to the main windows and tools of PPO. When an icon is selected, its background changes to orange as shown for several icons below.


yogurt_super_resources.csv - IBM ILOG Plant PowerOps
File Edit View Tools Window Help

What the icons do:

Displays the **Open Scenario** file browser window, allowing you to open a data file.

Saves the current, active scenario.

This tool lets you import data from Microsoft® Excel spreadsheet files.

This tool lets you export data to Microsoft Excel spreadsheet files.

Shows or hides the **Workspace View**.

Shows or hides the **KPI Comparison Panel**.

Shows or hides the **Checker**.

Shows or hides the **Console**.

Shows or hides the **Inspector**. See *The Inspector* for more information.

Shows or hides the **Changes to scenario** window, which allows you to undo and redo changes made to the Gantt chart.

Shows or hides the **Parameters** window which allows you to control how material and inventories appear in the Plan View. See *The Parameters window* for more information.

Starts Tableau enabling you to create a variety of reports regarding your data. See *Customizing report generation with Tableau* for more information.

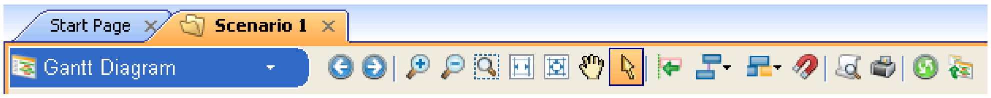
Optimizes the problem data to create a plan. See *Optimization and KPIs* for more information.

Allows you to undo or redo changes made to the **Gantt Diagram**.

Also refer to *The tool icons* on the *Plan View toolbar*.

Plan View toolbar

The toolbar for the **Gantt Diagram** view is shown below, with its many icons. The Plan View toolbar changes depending on which view is displayed. For example, **KPIs Summary** is a view-only page and so has very few tools.

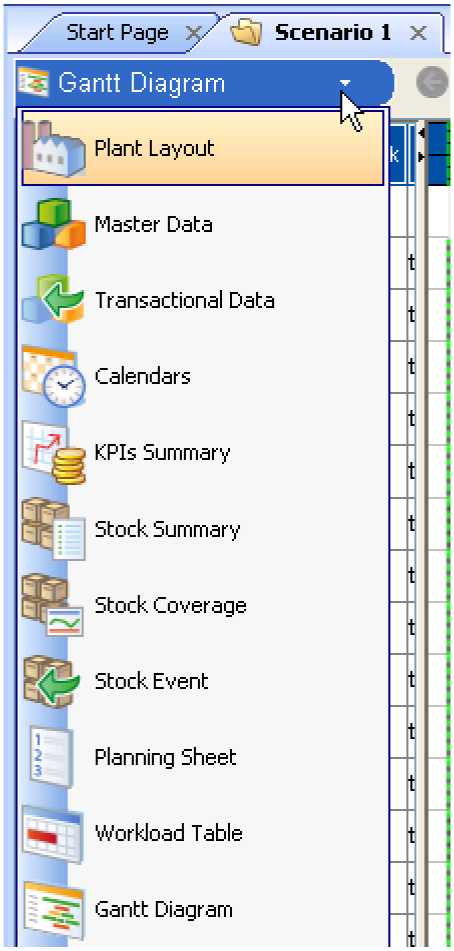


Above the toolbar is the title bar; the **Start Page** tab allows you to return to the initial starting page of PPO. Next to that are tabs for each of the open scenarios; here there is just one, **Scenario 1**.

Most of the icon tools on the **Gantt Diagram** view have the same or similar function when appearing on a different view.

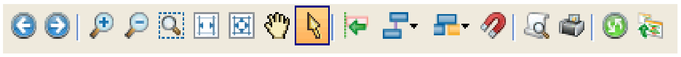
Gantt Diagram toolbar

The first selection available on the toolbar displays the view types bar. Selecting it gives you access to the various views of the plan: **Master Data**, **Calendars**, **KPIs Summary** and **Stock Summary**, for example.



The tool icons

Next on the toolbar are the many icons of the **Gantt Diagram** view.



Here’s what these tools do.

These arrows allow you to page forwards and backwards through your viewing history of the Plan Views for the current scenario. Only views which you have previously viewed are accessible through these navigation arrows.

These icons change the viewable window size on the Gantt chart.

The first three of these icons (from left to right) zoom in, zoom out, and allow you to draw a zoom expandable box on the Gantt chart (the box shape that you draw will expand to fill the Plan View window). The next icon fits the horizontal contents of the Plan View to the current window size, leaving the vertical contents unchanged. The last of these icons fits the horizontal contents too, and also centers the top of the Gantt chart in the current Plan View window size.

The pan tool; “grab” the chart and move to see other sections of the plan.

Activates the selection tool, allowing you to select activities and production orders on the **Gantt Diagram** to view information or move them.

Packs the activities to the left on the chart.

Activates magnetism, used as an aid to drag and drop operations performed on the **Gantt Diagram**. See *Repair extent, capacity and magnetism* for more information.

Tools that set the repair extent and capacity, which determine how PPO interprets your changes to the **Gantt Diagram**. See *Repair extent, capacity and magnetism* for more information.

Two icons to assist and control printing.

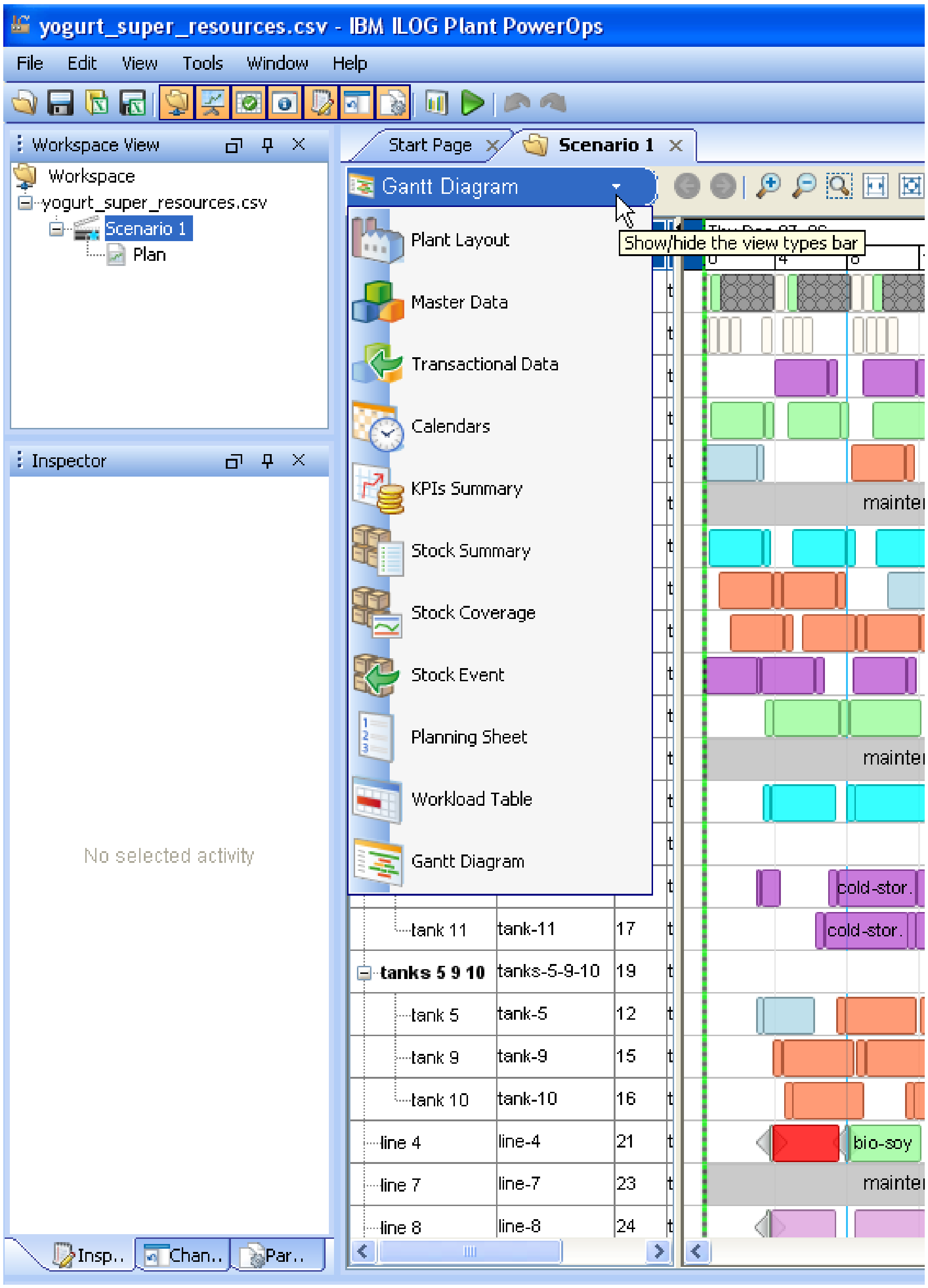
Refreshes the current view.

This tool lets you synchronize the **Gantt Diagram** chart with other views such as the **Calendars** and **Planning Workload**. Select this icon on each desired view, and when you scroll horizontally through the time scale on one view, the other synchronized views will remain correlated, making it very easy to compare data for a particular time. See *Splitting and synchronizing plan views* for more information.

Icons that control the view of a *single* plan. See *Splitting and synchronizing plan views* for more information.

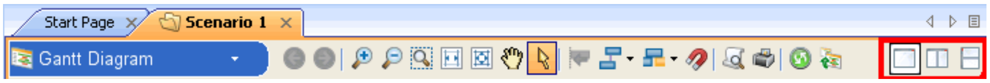
Accessing the plan views

After you optimize a scenario, PPO presents the solution data in a number of plan views. By default, the **Gantt Diagram** displays first, but you can access the other views by using the *view types* bar; click **Gantt Diagram** as shown in this image to reveal the other plan views available to you.


yogurt_super_resources.csv - IBM ILOG Plant PowerOps
File
Edit
View
Tools
Window
Help
Workspace View
Workspace
yogurt_super_resources.csv
Scenario 1
Plan
Inspector
No selected activity
Insp..
Chan..
Par..
Start Page
Scenario 1
Gantt Diagram
Show/hide the view types bar
Plant Layout
Master Data
Transactional Data
Calendars
KPIs Summary
Stock Summary
Stock Coverage
Stock Event
Planning Sheet
Workload Table
Gantt Diagram
tank 11
tank-11
17
tanks 5 9 10
tanks-5-9-10
19
tank 5
tank-5
12
tank 9
tank-9
15
tank 10
tank-10
16
line 4
line-4
21
line 7
line-7
23
line 8
line-8
24
mainte
cold-stor.
bio-soy

Splitting and synchronizing plan views

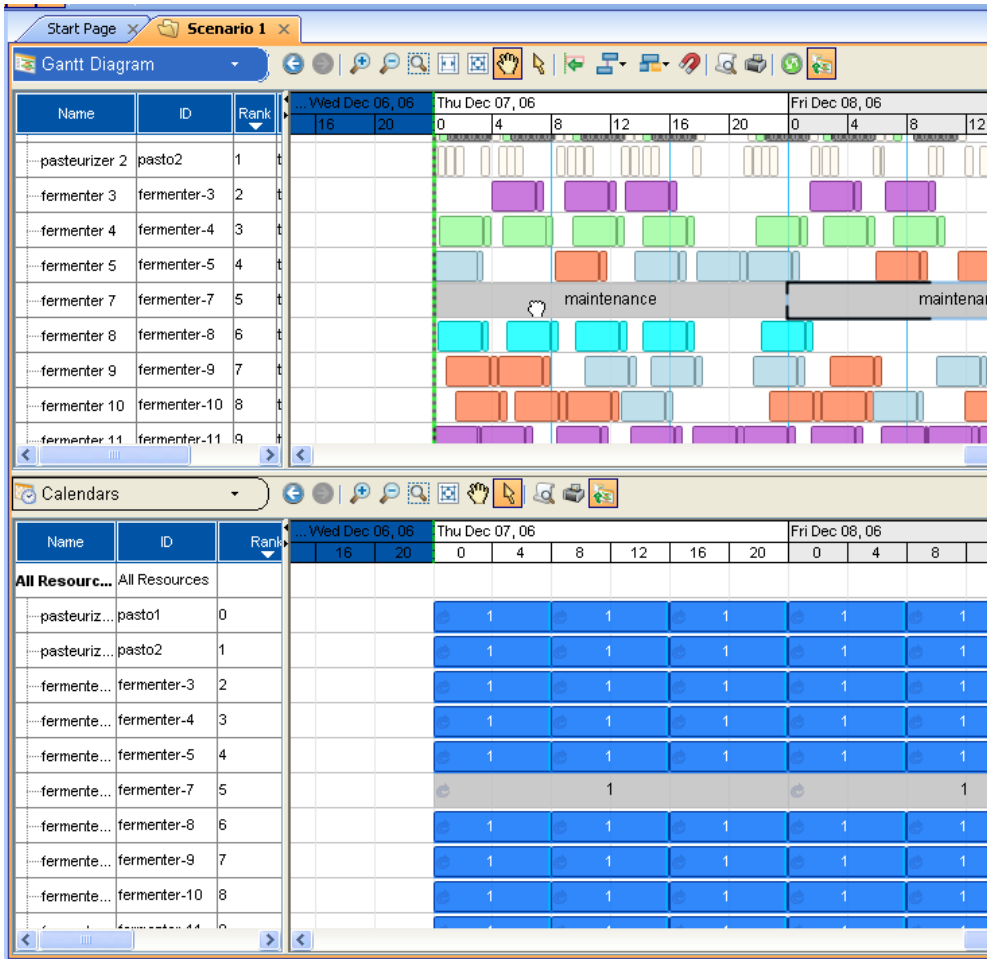
By default you can only see one of the plan views at a time. Undoubtedly you'll eventually want to look at two of the plan views at once to compare data. Use the split view controls (highlighted below in red) to achieve this.



Once two views within a scenario are displayed, there are several techniques to help control the data you see. One method is to *filter* the data; this is discussed in *Using the Filter tool on the Master and Transactional Data tables*.

Another technique is to synchronize views; this “locks” two or more views together such that when you pan or move one view along its time line, the other view moves in synchronization with the first. The synchronization tool is located on the toolbar, and when activated has an orange background as shown here:

The following image shows the synchronization tool activated (on toolbar, far right), and the pan tool selected (*hand* icon on toolbar, orange background). The pan tool has been used to move the **Gantt Diagram** to the right, and the **Calendars** view has moved with it, revealing the start min area for both views.



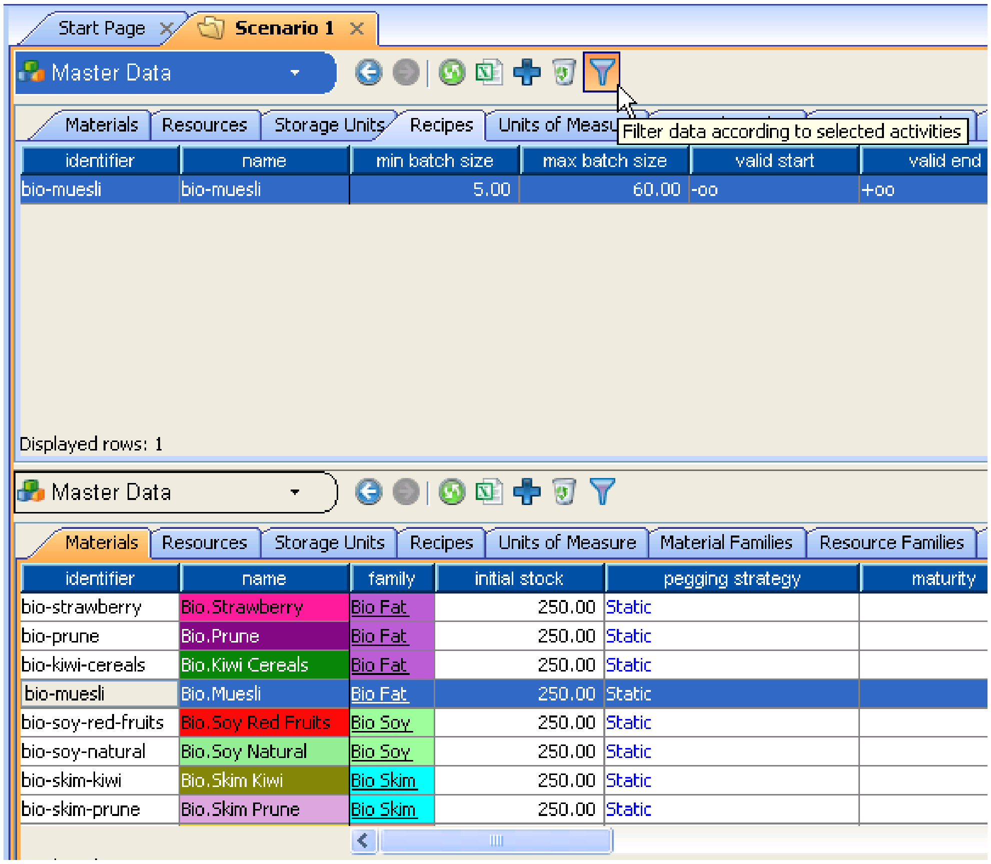
Name	ID	Rank
pasteurizer 2	pasto2	1
fermenter 3	fermenter-3	2
fermenter 4	fermenter-4	3
fermenter 5	fermenter-5	4
fermenter 7	fermenter-7	5
fermenter 8	fermenter-8	6
fermenter 9	fermenter-9	7
fermenter 10	fermenter-10	8
fermenter 11	fermenter-11	9

Name	ID	Rank
All Resourc...	All Resources	
pasteuriz...	pasto1	0
pasteuriz...	pasto2	1
fermente...	fermenter-3	2
fermente...	fermenter-4	3
fermente...	fermenter-5	4
fermente...	fermenter-7	5
fermente...	fermenter-8	6
fermente...	fermenter-9	7
fermente...	fermenter-10	8

Using the Filter tool on the Master and Transactional Data tables

When displaying the **Master Data** or **Transactional Data** tables, you can filter the displayed data by various criteria, including by materials, recipes and resources.

For example, the following image shows two **Master Data** views, the lower of which is focused on the **Materials** tab with the material **bio-muesli** selected. The upper view is focused on the **Recipes** tab, where the user had selected the filter icon with the result that the only recipe displayed is the one associated with the bio-muesli material.



identifier	name	min batch size	max batch size	valid start	valid end
bio-muesli	bio-muesli	5.00	60.00	-oo	+oo

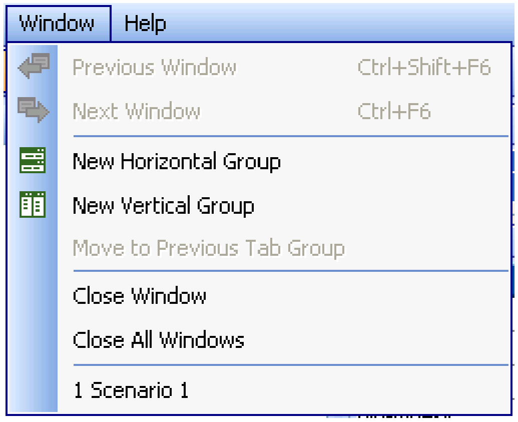
Displayed rows: 1

identifier	name	family	initial stock	pegging strategy	maturity
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	
bio-prune	Bio.Prune	Bio Fat	250.00	Static	
bio-kiwi-cereals	Bio.Kiwi Cereals	Bio Fat	250.00	Static	
bio-muesli	Bio.Muesli	Bio Fat	250.00	Static	
bio-soy-red-fruits	Bio.Soy Red Fruits	Bio Soy	250.00	Static	
bio-soy-natural	Bio.Soy Natural	Bio Soy	250.00	Static	
bio-skim-kiwi	Bio.Skim Kiwi	Bio Skim	250.00	Static	
bio-skim-prune	Bio.Skim Prune	Bio Skim	250.00	Static	

The filter tool also works in combination with other views; for example, if you select an object on the **Gantt Diagram**, and then select the filter tool on the **Master** or **Transactional Data** views, then only the data rows associated with the selected items will continue to display.

Comparing multiple scenarios

One of the notable features of PPO is the ability to create multiple scenarios that have been optimized from different data sets, and keep these scenarios available simultaneously in the PPO workspace. To compare two scenarios, use the **New Horizontal Group** and **New Vertical Group** tools on the **Window** menu.

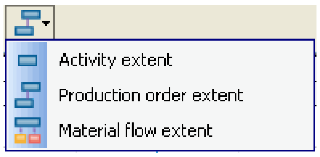


Either command allows you to simultaneously display views from both scenarios; you can compare the **Gantt Diagram** charts of both, the **Workload Tables**, **Stock Coverage** views, and so forth. You can even display two views within each scenario, as described in *Splitting and synchronizing plan views*.

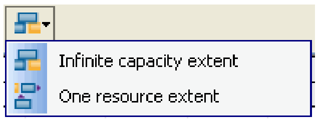
Repair extent, capacity and magnetism

When you edit a plan with a drag and drop action in the **Gantt Diagram** view, it is possible to interpret that action in a variety of ways due to the constraints and material flows involved with the activity or production order that you are editing. *Repair extent* and *repair capacity* solve that problem. They specify to Plant PowerOps exactly how to interpret your changes. Additionally, *magnetism* aids in specifying your target location on the Gantt chart. Repair extent and capacity are set using the following icons on the **Gantt Diagram** toolbar. The left icon allows you to set the repair extent, and the right one sets the resource capacity of the repair.

The following image shows the choices available for repair extent (some nearby icons have been removed from the image for clarity). You can choose an **Activity extent**, which means that only the activities that are actually selected on the Gantt will have their constraints enforced; this means that duration and calendar constraints are respected, but not precedence constraints. **Production order extent** means that all activities that belong to the production order(s) of the selected activities will have their constraints enforced; this adds enforcement of precedence constraints for those activities. **Material flow extent** adds enforcement of the pegging constraints for the clusters of production orders of the selected activities.



The following image shows the choices available for repair capacity. **Infinite capacity extent** means that resource capacity constraints involved with the edit are ignored. **One resource extent** means that the capacity constraints of the primary resource of the selected activities are enforced, and overlapping activities are changed to eliminate the conflict. To use this selection, all selected activities should have the same primary resource, and this resource should have a unary capacity.



The magnetism tool is provided by this icon: . Magnetism is an “attraction” between the activity you are moving and the closest graphical object near to where you place it. This helps to eliminate any empty time slot on the chart, as the objects “snap” together.. When magnetism is activated, the icon has an orange background.

You can decide to include shifts and breaks in the magnetism move on the **Tools —> Options —> Advanced Parameters** window.

Be aware that any change you make to a plan is lost if you optimize the problem again, unless you firm or fix your changes.

Stock Coverage view

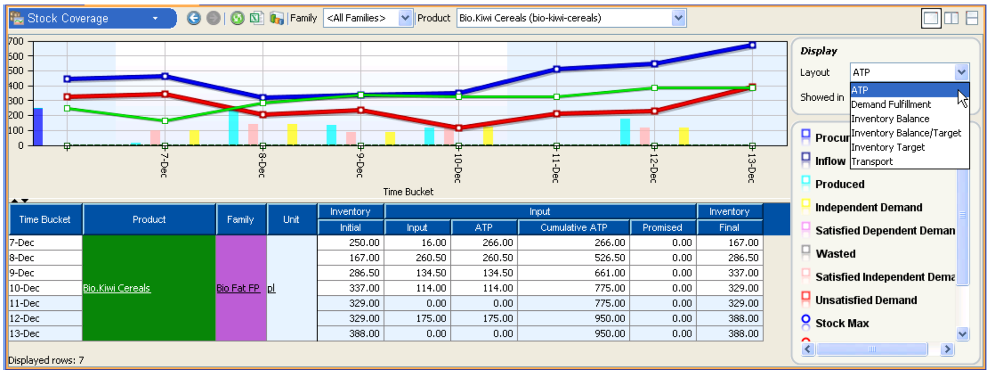
The **Stock Coverage** view has selectable **Layouts**, as shown in the figure below: **ATP**, **Demand Fulfillment**, **Inventory Balance/Target** (default), and so forth. Each layout changes the visible data columns in the table below the graph, and is tailored to specific needs such as inventory management, transportation, ATP (available to promise) calculations, and so forth.

You can also temporarily add or remove columns from a layout with the contextual menu; right-click and on the menu select **Visibility**. For long-term custom changes, you can edit these layouts and create your own. Each layout is stored in the <*PPOInstallDirectory*>`/data/gui/table` directory in a file named in the manner `stockcoveragedisplayxxxx.xml`. See *Customizing and Extending PPO* for more information.

A description of some standard layouts:

- **ATP**: This layout is for the available-to-promise use case and shows the ATP, Promised and Cumulative ATP columns. See *Production planning simulations* for a use example.
- **Demand Fulfillment**: This layout shows how much of the demand due in a bucket is satisfied (over all buckets included in its delivery window).
- **Inventory Balance**: This layout focuses on the different terms of the material balance equation: inputs and outputs are detailed.
- **Inventory Target**: This layout focuses on the inventory target corridor and alerts when the inventory goes below or above the corridor.

Regarding inventory consumption by demand and how this affects this view, there is automatic repegging to demand at each interaction that enforces the constraints and may not satisfy the whole demand. The stock displayed can not be negative, nor can immature or obsolete stock be consumed by the demand. A red bar indicates the unsatisfied demand. A gray bar indicates the obsolete inventory thrown away at expiration time. The inflow bar indicates the amount of missing intermediate material. The automatic demand repegging is based on a linear optimizer taking the scheduling weights in its objective function. It pegs the procurements and production orders to the demand taking the scheduling solution into account. If scheduling is not required by the optimization profile, and for time periods past the scheduling horizon, the planned productions coming from the planning solution are also pegged to the demand.



Time Bucket	Product	Family	Unit	Inventory	Input				Inventory
				Initial	Input	ATP	Cumulative ATP	Promised	Final
7-Dec				250.00	16.00	266.00	266.00	0.00	167.00
8-Dec				167.00	260.50	260.50	526.50	0.00	286.50
9-Dec				286.50	134.50	134.50	661.00	0.00	337.00
10-Dec	Bio.Kiwi Cereals	Bio Fat FP	pl	337.00	114.00	114.00	775.00	0.00	329.00
11-Dec				329.00	0.00	0.00	775.00	0.00	329.00
12-Dec				329.00	175.00	175.00	950.00	0.00	388.00
13-Dec				388.00	0.00	0.00	950.00	0.00	388.00

Displayed rows: 7

Column descriptions

Inventory is computed for each time bucket as follows:

- Final inventory = Initial inventory + Input - Output
- Input = Procured + Produced + Inflow
- Output = Satisfied Independent Demand + Dependent Demand + Waste

Demand fulfillment is determined as follows:

- The demand fulfillment gives the percentage of satisfaction of the independent demand: 100*(Independent Demand - Unsatisfied Demand)/ Independent Demand
- The independent demand indicates the amount of material requested per bucket. The demand is assigned to the bucket containing the due date -1, or the delivery end max -1 if the demand has no due date. Note that a demand may not necessarily be satisfied in its due bucket.
- The unsatisfied demand gives the part of the independent demand due in a bucket that is left unsatisfied (over the whole delivery window).

Inventory targets information:

- The stock min and stock max give the lower bound and the upper bound of the inventory target corridor.
- This may be expressed in quantity (display unit of the selected material) or in days of supply, which is the number of days of demand to cover with the inventory at the end of a bucket.
- The **Alert** column warns if the final inventory is above or below the inventory target corridor.

Available-to-promise is calculated as follows:

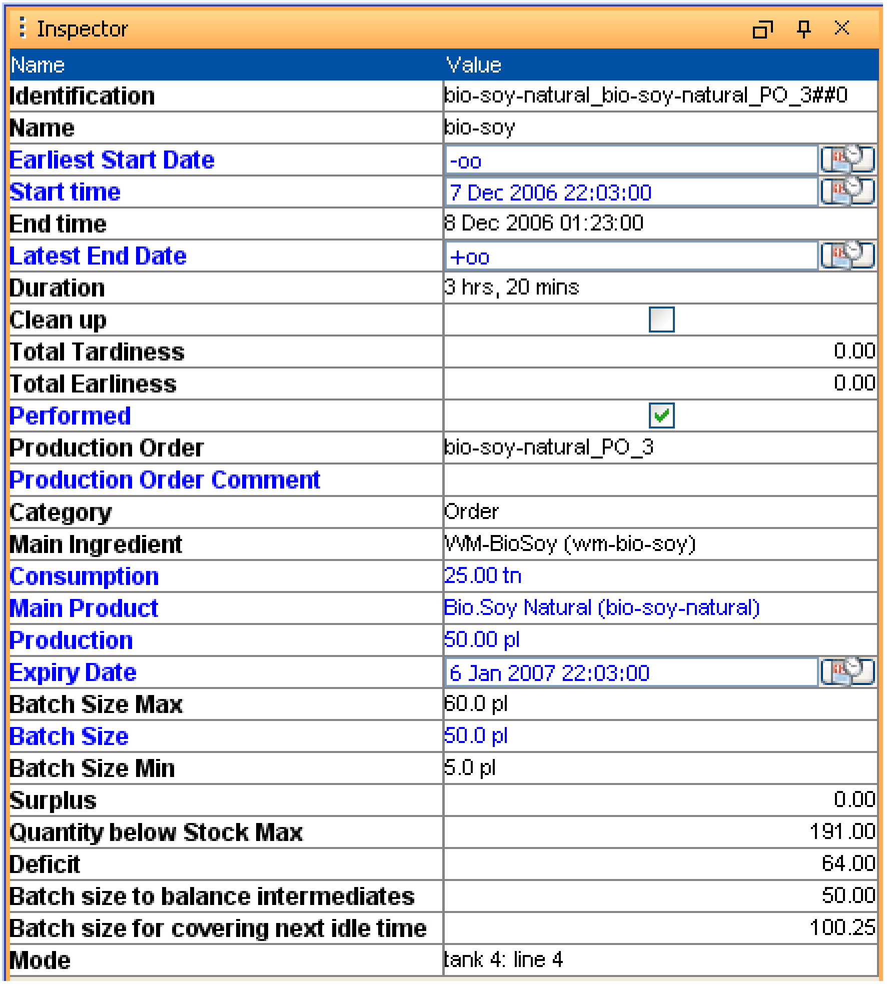
- The promised values is the part of the independent demand that has been already promised to customers. The rest the of the independent demand is a forecast.
- Available-to-promise gives the part of the production of a bucket that is still available to promise.
- The cumulative ATP gives the total inventory available to promise in a given bucket.

The **In Transit** inventory gives the amount of material in transit at the end of a bucket and is expected to arrive in future time buckets.

The Inspector

The **Inspector** is a tool used in conjunction with the **Gantt Diagram**. It provides details about items selected on the Gantt, including setup activities, production orders, and maintenance and cleanup operations. The Inspector can be displayed or hidden through the **Tools —> Inspector** selection on the menu bar.

The following image shows a typical example of the **Inspector** when an activity has been selected on the **Gantt Diagram**.



Name	Value
Identification	bio-soy-natural_bio-soy-natural_PO_3##0
Name	bio-soy
Earliest Start Date	-oo
Start time	7 Dec 2006 22:03:00
End time	8 Dec 2006 01:23:00
Latest End Date	+oo
Duration	3 hrs, 20 mins
Clean up	☐
Total Tardiness	0.00
Total Earliness	0.00
Performed	✓
Production Order	bio-soy-natural_PO_3
Production Order Comment	
Category	Order
Main Ingredient	WM-BioSoy (wm-bio-soy)
Consumption	25.00 tn
Main Product	Bio.Soy Natural (bio-soy-natural)
Production	50.00 pl
Expiry Date	6 Jan 2007 22:03:00
Batch Size Max	60.0 pl
Batch Size	50.0 pl
Batch Size Min	5.0 pl
Surplus	0.00
Quantity below Stock Max	191.00
Deficit	64.00
Batch size to balance intermediates	50.00
Batch size for covering next idle time	100.25
Mode	tank 4: line 4

The Inspector typically includes information from both a selected activity and from the production order to which it belongs. The **Name** is the name of the selected activity, and **Identification** reveals the associated production order. Fields colored in blue indicate data you can edit in the Inspector; for example, you can add a **Production Order Comment** or edit dates and sizes.

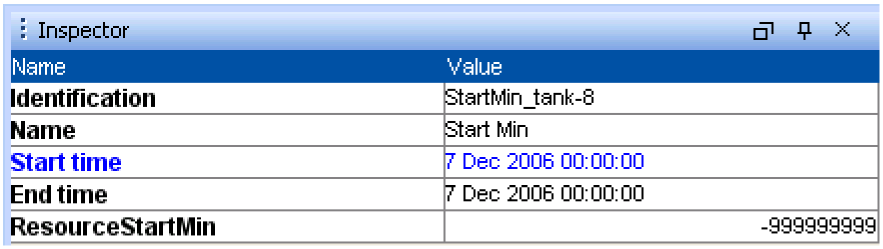
You can adjust the batch size of a production order by changing the quantity produced or the quantity consumed directly in the Inspector. Moreover, the batch size, consumption, and production can be specified in different units if enabled in the model (secondary units for material declared). In the image above, the consumption is specified in tons, and the production and batch size is in pallets. The batch size is updated automatically when fields are edited. The modification of the batch size may influence the size of all activities on the production order and the production/consumption of intermediates and finished products.

The **Expiry Date** is the expiration date of the material. A default expiration date is computed from the shelf life of the product produced by a production order. The plant scheduler can override this expiration date. When moving an order, the expiration date is recomputed automatically from the shelf life data of the corresponding product.

Batch size to balance intermediates is the proposed batch size of the selected production order to best utilize intermediate products.

Batch size for covering next idle time is the proposed batch size to utilize the resource idle time that is located to the right of the selected production order on the Gantt Diagram, for the next production order on the same resource.

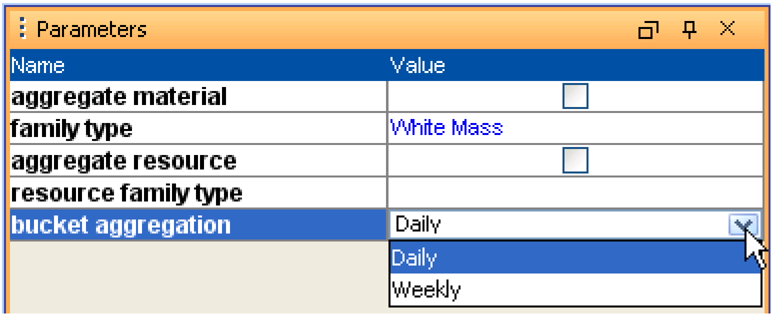
The following image shows the Inspector displaying start min information for a resource; this is a convenient way to change the start min value.



Name	Value
Identification	StartMin_tank-8
Name	Start Min
Start time	7 Dec 2006 00:00:00
End time	7 Dec 2006 00:00:00
ResourceStartMin	-999999999

The Parameters window

The **Parameters** window allows you to control visibility aspects on the stock, inventory, planning and workload charts. For example, in the following image the user can select from displaying data according to **Daily** or **Weekly** time buckets. The time buckets must be defined in the data model to be available here.



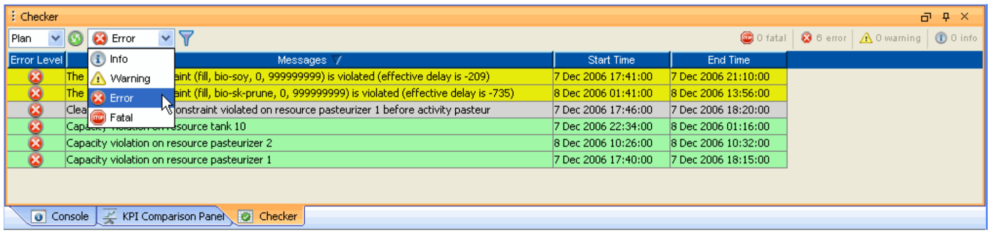
Name	Value
aggregate material	☐
family type	White Mass
aggregate resource	☐
resource family type	
bucket aggregation	Daily

You can also display data organized according to material type and resource family type. For example, select the **aggregate material** check box, then use the combo box in the **family type** value field to select the appropriate material family.

Using the checker

The **Checker** is a tool that you can use to test your **Plan** and **Model** data. You can keep the checker open while you edit a data view or the Gantt chart to ensure that your changes don't violate constraints or cause other problems. The **Checker** synchronizes with the **Gantt Diagram** view to help you locate problems

To display the **Checker**, press the **F12** key, or select **Tools** --> **Show Checker** on the *The menu bar*, or select the checker icon. The **Checker** window displays in the bottom portion of the GUI in the same space as the **Console** and **KPI Comparison Panel**.



Error Level	Messages	Start Time	End Time
❌	The aint (fill, bio-soy, 0, 999999999) is violated (effective delay is -209)	7 Dec 2006 17:41:00	7 Dec 2006 21:10:00
❌	The aint (fill, bio-sk-prune, 0, 999999999) is violated (effective delay is -735)	8 Dec 2006 01:41:00	8 Dec 2006 13:56:00
❌	Clea onstraint violated on resource pasteurizer 1 before activity pasteur	7 Dec 2006 17:46:00	7 Dec 2006 18:20:00
❌	Capa source tank 10	7 Dec 2006 22:34:00	8 Dec 2006 01:16:00
❌	Capacity violation on resource pasteurizer 2	8 Dec 2006 10:26:00	8 Dec 2006 10:32:00
❌	Capacity violation on resource pasteurizer 1	7 Dec 2006 17:40:00	7 Dec 2006 18:15:00

In the image above, messages for the **Plan** are displayed; you can choose to view messages for the **Model** if desired. The user is selecting **Error** which means that only messages of at least that level will display in the **Checker** (informational and warning messages would be filtered out). This is a useful filtering mechanism as some plans or models generate many messages.

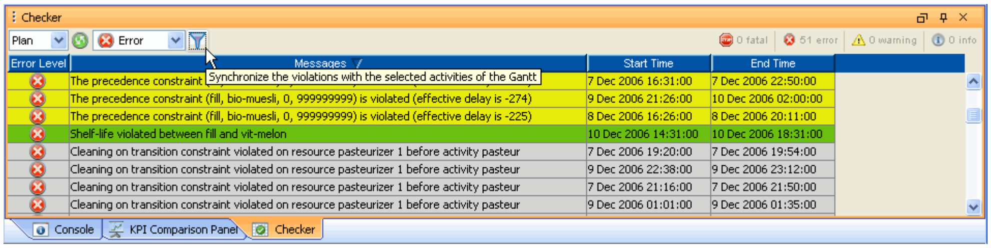
If you select the **Refresh** icon, a test is performed for data plan and model integrity. The refresh checks both the **Plan** and the **Model**, and will switch views if one has errors and the other does not.

Filtering error messages and synchronizing with the Gantt

As already mentioned, you can filter the messages displayed in the **Checker** window according to relevancy to the **Plan** or to the **Model**, and according to message severity (informational, warning, error, or fatal).

Once the messages are narrowed down to a manageable number, the simplest method of using the **Checker** is to simply select the message in the **Checker**; the associated activity object and production order will be highlighted in the **Gantt Diagram**.

You can also “filter” the messages by selecting an object on the **Gantt Diagram**, then selecting the synchronize icon in the checker, as shown in the following image. The result is that only messages associated with the selected Gantt object will appear in the **Checker**.

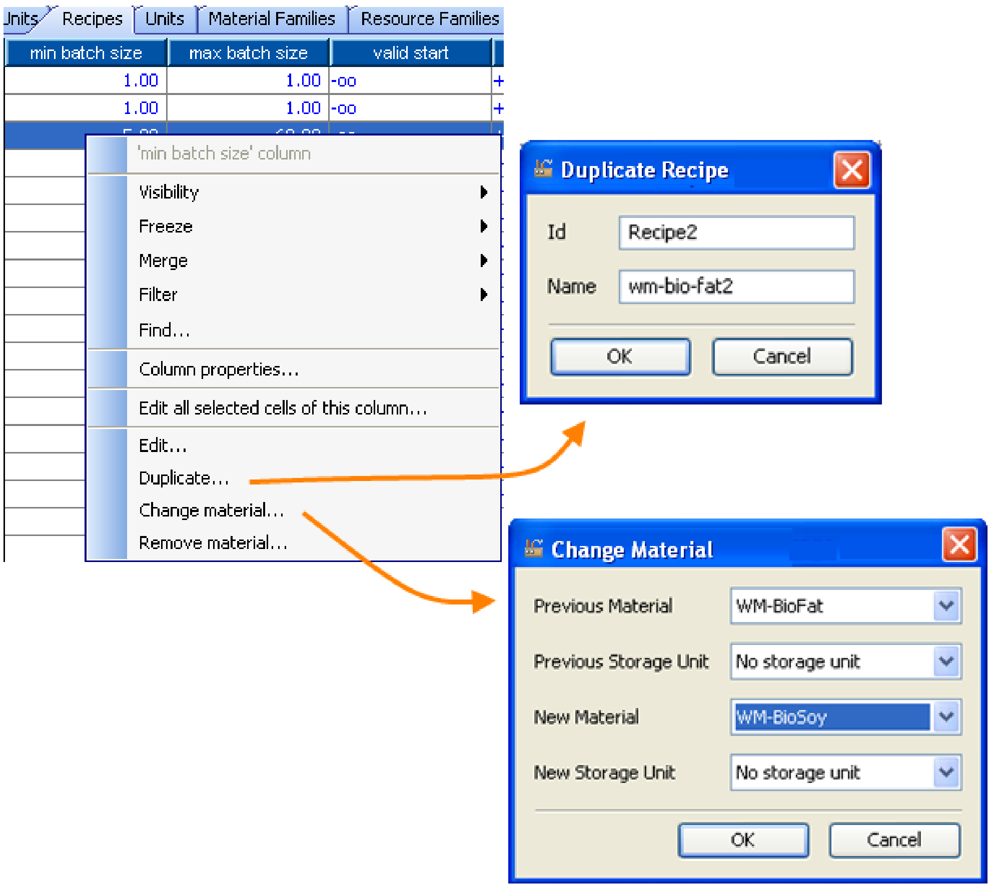


Error Level	Messages	Start Time	End Time
⊗	The precedence constraint	7 Dec 2006 16:31:00	7 Dec 2006 22:50:00
⊗	The precedence constraint (fill, bio-muesli, 0, 999999999) is violated (effective delay is -274)	9 Dec 2006 21:26:00	10 Dec 2006 02:00:00
⊗	The precedence constraint (fill, bio-muesli, 0, 999999999) is violated (effective delay is -225)	8 Dec 2006 16:26:00	8 Dec 2006 20:11:00
⊗	Shelf-life violated between fill and vit-melon	10 Dec 2006 14:31:00	10 Dec 2006 18:31:00
⊗	Cleaning on transition constraint violated on resource pasteurizer 1 before activity pasteur	7 Dec 2006 19:20:00	7 Dec 2006 19:54:00
⊗	Cleaning on transition constraint violated on resource pasteurizer 1 before activity pasteur	9 Dec 2006 22:38:00	9 Dec 2006 23:12:00
⊗	Cleaning on transition constraint violated on resource pasteurizer 1 before activity pasteur	7 Dec 2006 21:16:00	7 Dec 2006 21:50:00
⊗	Cleaning on transition constraint violated on resource pasteurizer 1 before activity pasteur	9 Dec 2006 01:01:00	9 Dec 2006 01:35:00

Remember to deselect the synchronize icon when finished.

Copying an existing recipe

Recipes can become quite complex in a modern facility. Suppose you have a very intricate process already modeled, and need a new recipe that is identical except for the material produced, production time, or storage unit. Using the contextual menu on the **Recipes** tab, it's easy to model this by copying the first recipe and then making the changes.



min batch size	max batch size	valid start
1.00	1.00	-oo
1.00	1.00	-oo

Duplicate Recipe

Id	Recipe2
Name	wm-bio-fat2

Change Material

Previous Material	WM-BioFat
Previous Storage Unit	No storage unit
New Material	WM-BioSoy
New Storage Unit	No storage unit

Database usage and connectivity

This describes the supported databases of PPO and their usage; for advanced topics see *Database customization*.

In this section

Supported databases
Several databases are supported.

Basic database connectivity
Basic use of a database in the GUI of PPO.

Adding custom database connectivity
Describes how to streamline interactions with the database.

Supported databases

This is the list of supported databases for PPO:

- Microsoft® Office Access® as a file-based database directly in the GUI (You can load and save PPO scenarios in Microsoft Access format using **File** > **Open scenario** and **File** > **Save scenario**. Microsoft Access is supported only on the Microsoft Windows® 32-bit platform.)
- Oracle® Database 10g and 11g.
- Microsoft SQL Server® 2000 and 2005.

Basic database connectivity

Basic use of a database in the GUI of PPO.

In this section

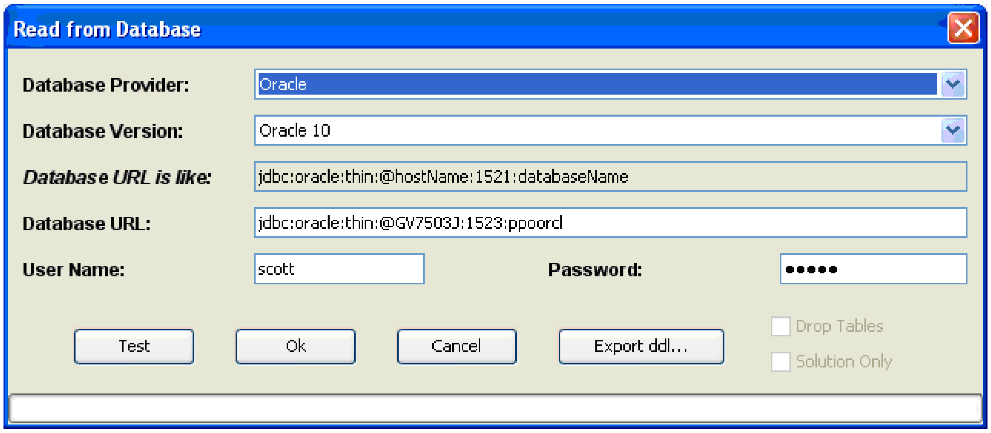
Opening a database in PPO
Describes the Open Scenario from Database dialog box.

Automatic bucket generation
Time buckets can be automatically created based on model data.

Save scenario in a database
Explains options while saving.

Opening a database in PPO

To open a PPO scenario from a database, select **File > Open Scenario from Database** to connect to the database server.


Read from Database

Database Provider: Oracle

Database Version: Oracle 10

Database URL is like: jdbc:oracle:thin:@hostName:1521:databaseName

Database URL: jdbc:oracle:thin:@GV7503J:1523:ppoorcl

User Name: scott

Password: ●●●●●

Choose the database provider in the first combo-box.

Choose one of the supported database versions in the second combo-box.

The *Database URL is Like* text field contains an example of proper connection string syntax. Follow that example and enter your data in the **Database URL** field.

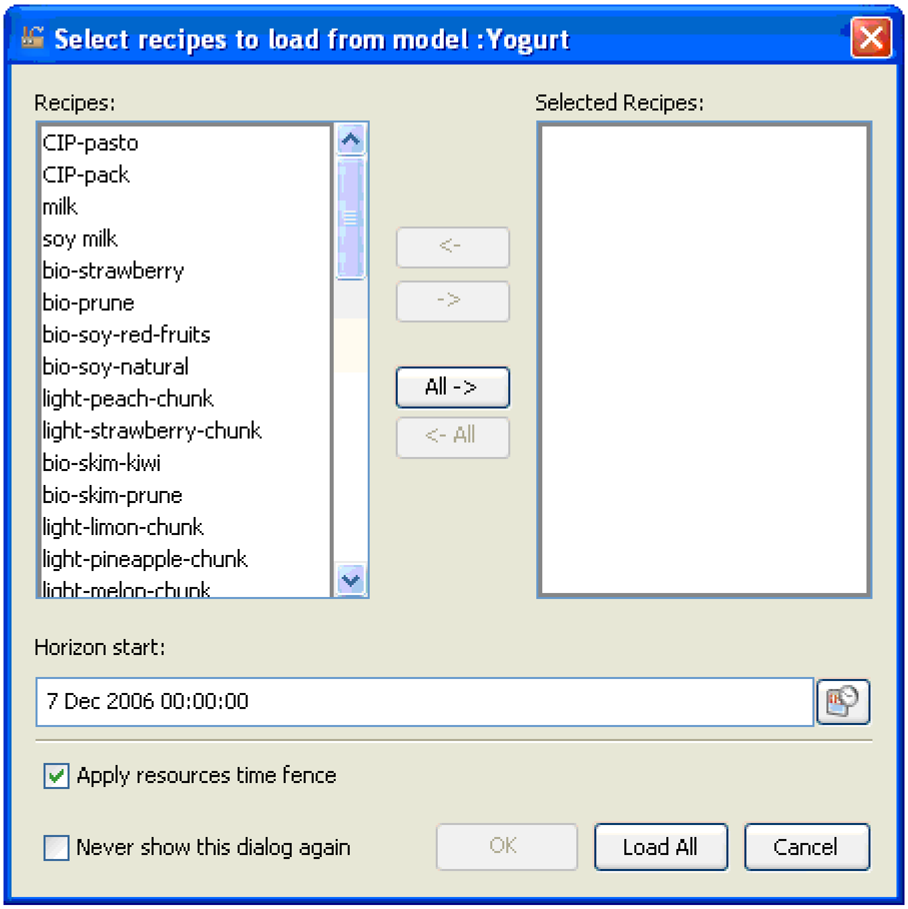
Enter the user name and the password.

You can test the connection to the database by pressing the **Test** button

Pressing **Ok** will load the scenario stored in the database.

Loading submodels from database

When loading a model from a database, a window displays allowing the user to select a set of recipes to load (a submodel):



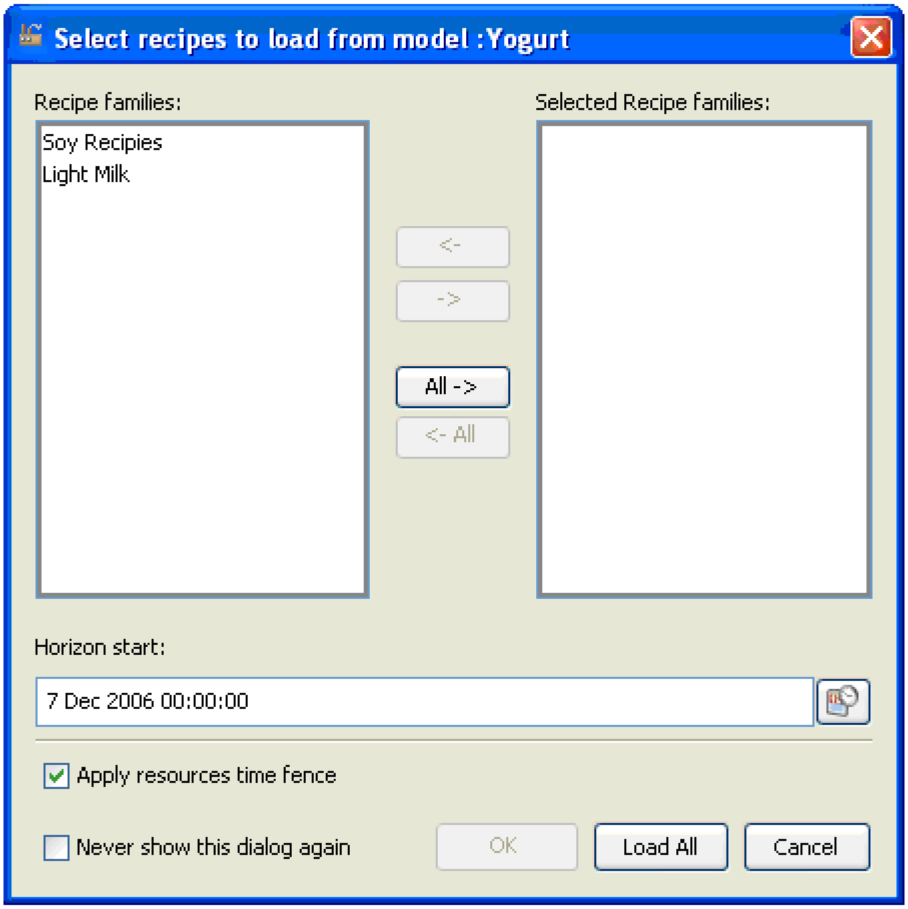
CIP-pasto
CIP-pack
milk
soy milk
bio-strawberry
bio-prune
bio-soy-red-fruits
bio-soy-natural
light-peach-chunk
light-strawberry-chunk
bio-skim-kiwi
bio-skim-prune
light-limon-chunk
light-pineapple-chunk
light-melon-chunk

Horizon start:

7 Dec 2006 00:00:00

- [x] Apply resources time fence
- [ ] Never show this dialog again

If the model contains records in the table PPO_RECIPE_FAMILY, the left-hand column contains a list of recipe families rather than individual recipes as shown in this image:


Select recipes to load from model :Yogurt
Recipe families:
Selected Recipe families:
<-
->
All ->
<- All
OK
Load All
Cancel

Soy Recipies
Light Milk

Horizon start: 7 Dec 2006 00:00:00

[✓] Apply resources time fence

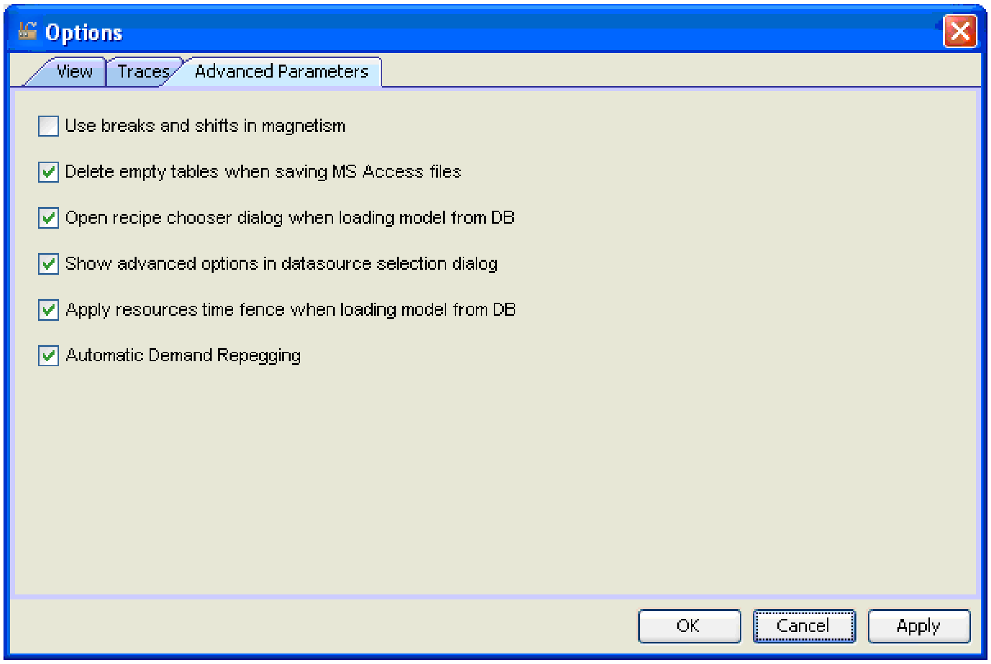
[ ] Never show this dialog again

When a submodel is loaded, you can run the engine and save the solution back into the database. In this case you have a partial solution of the whole model.

Warning: You must be careful when designing your submodels to be loaded and solved with PPO. To ensure a coherent global solution you must ensure that your submodels do NOT share resources. When a submodel is loaded, PPO is not able to take into account the solution (planned/scheduled activities) on resources shared by the other recipes (that is, recipes which are not part of the loaded submodel).

In the recipe selector window, you can set the **Horizon start** of your model and choose to apply the time fence of the resources. This time fence is defined in the column TIME_FENCE of the PPO_RESOURCE table.

You can stop PPO from displaying this window again with the **Never show this dialog again** checkbox. You can reactivate this window in the **Advanced Parameters** tab of the **Options** panel (menu **Tools->Options**), shown below.

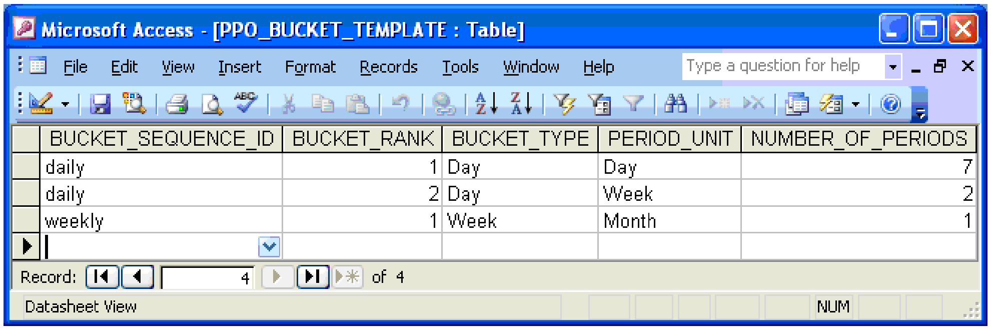


- [ ] Use breaks and shifts in magnetism
- [✓] Delete empty tables when saving MS Access files
- [✓] Open recipe chooser dialog when loading model from DB
- [✓] Show advanced options in datasource selection dialog
- [✓] Apply resources time fence when loading model from DB
- [✓] Automatic Demand Repegging

Automatic bucket generation

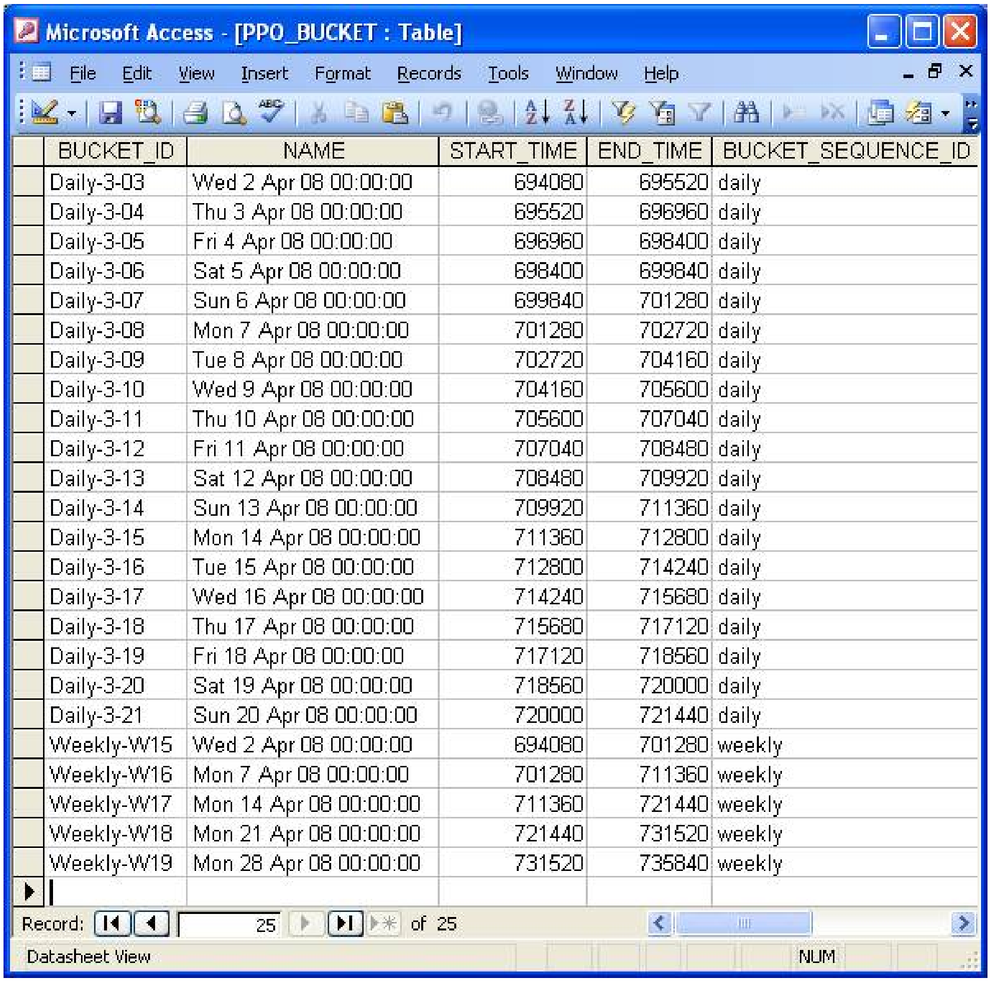
When loading a model from a database (Microsoft® Office Access® file or database server like Oracle® or SQL Server®) and the model contains records in the table PPO_BUCKET_TEMPLATE, then PPO generates bucket instances starting from the Horizon Start of your model based on the TIME_UNIT of your model. The horizon is set in the recipe selector window or with the column START_MIN of the PPO_MODEL table.

As an example take a model with a time unit of 60 seconds and the following PPO_BUCKET_TEMPLATE table:


Microsoft Access - [PPO_BUCKET_TEMPLATE : Table]
File Edit View Insert Format Records Tools Window Help
Type a question for help
Record: 4 of 4
Datasheet View
NUM

BUCKET_SEQUENCE_ID	BUCKET_RANK	BUCKET_TYPE	PERIOD_UNIT	NUMBER_OF_PERIODS
daily	1	Day	Day	7
daily	2	Day	Week	2
weekly	1	Week	Month	1

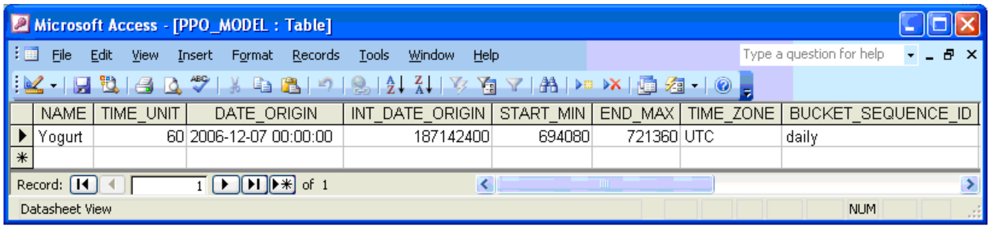
Then PPO will generate the following time buckets if your horizon start is April 2nd 2008:


Microsoft Access - [PPO_BUCKET : Table]

BUCKET_ID	NAME	START_TIME	END_TIME	BUCKET_SEQUENCE_ID
Daily-3-03	Wed 2 Apr 08 00:00:00	694080	695520	daily
Daily-3-04	Thu 3 Apr 08 00:00:00	695520	696960	daily
Daily-3-05	Fri 4 Apr 08 00:00:00	696960	698400	daily
Daily-3-06	Sat 5 Apr 08 00:00:00	698400	699840	daily
Daily-3-07	Sun 6 Apr 08 00:00:00	699840	701280	daily
Daily-3-08	Mon 7 Apr 08 00:00:00	701280	702720	daily
Daily-3-09	Tue 8 Apr 08 00:00:00	702720	704160	daily
Daily-3-10	Wed 9 Apr 08 00:00:00	704160	705600	daily
Daily-3-11	Thu 10 Apr 08 00:00:00	705600	707040	daily
Daily-3-12	Fri 11 Apr 08 00:00:00	707040	708480	daily
Daily-3-13	Sat 12 Apr 08 00:00:00	708480	709920	daily
Daily-3-14	Sun 13 Apr 08 00:00:00	709920	711360	daily
Daily-3-15	Mon 14 Apr 08 00:00:00	711360	712800	daily
Daily-3-16	Tue 15 Apr 08 00:00:00	712800	714240	daily
Daily-3-17	Wed 16 Apr 08 00:00:00	714240	715680	daily
Daily-3-18	Thu 17 Apr 08 00:00:00	715680	717120	daily
Daily-3-19	Fri 18 Apr 08 00:00:00	717120	718560	daily
Daily-3-20	Sat 19 Apr 08 00:00:00	718560	720000	daily
Daily-3-21	Sun 20 Apr 08 00:00:00	720000	721440	daily
Weekly-W15	Wed 2 Apr 08 00:00:00	694080	701280	weekly
Weekly-W16	Mon 7 Apr 08 00:00:00	701280	711360	weekly
Weekly-W17	Mon 14 Apr 08 00:00:00	711360	721440	weekly
Weekly-W18	Mon 21 Apr 08 00:00:00	721440	731520	weekly
Weekly-W19	Mon 28 Apr 08 00:00:00	731520	735840	weekly

Record: 25 of 25

Datasheet View


Microsoft Access - [PPO_MODEL : Table]

NAME	TIME_UNIT	DATE_ORIGIN	INT_DATE_ORIGIN	START_MIN	END_MAX	TIME_ZONE	BUCKET_SEQUENCE_ID
Yogurt	60	2006-12-07 00:00:00	187142400	694080	721360	UTC	daily

Record: 1 of 1

Datasheet View

In this case, for the daily bucket sequence, PPO generates seven buckets of type **Day** (from **Daily-3-03** to **Daily-3-09**) for the bucket rank **1**; and 12 buckets of type **Day** (from **Daily -3-10** to **Daily-3-21**) for the rank **2**. For the weekly bucket sequence, PPO will generate five buckets; the last four are one week length while the first one is five days in length (from Wednesday to Sunday).

Note: The NAME field of the PPO_BUCKET table, above, has been generated in a typical day-month format. The values in the BUCKET_ID field denote the end of the time bucket, with the first month of the year represented by zero and the first day of a month starting at one.

By default the week buckets start on Monday. To change the starting day of the week buckets you can use the method `ilog.plant.persistence.loader.DefaultBucketGenerator.setBeginningOfTheWeek(int)`.

Save scenario in a database

Select **File** > **Save Scenario in Database** to connect to a database server and save a PPO Scenario.

Choose the database provider in the first combo-box, and choose one of the supported database versions in the second combo-box.

The *Database URL is Like* text field contains an example of proper connection string syntax. Follow that example and enter your data in the **Database URL** field.

Enter the user name and the password. You can test the connection to the database by pressing the **Test** button.

Pressing **Ok** will save the scenario in the selected database.

You can check the **Drop Tables** option to drop PPO tables and/or create the tables in the database, and you can also export the PPO database description to a `.ddl` file using the **Export ddl...** button.

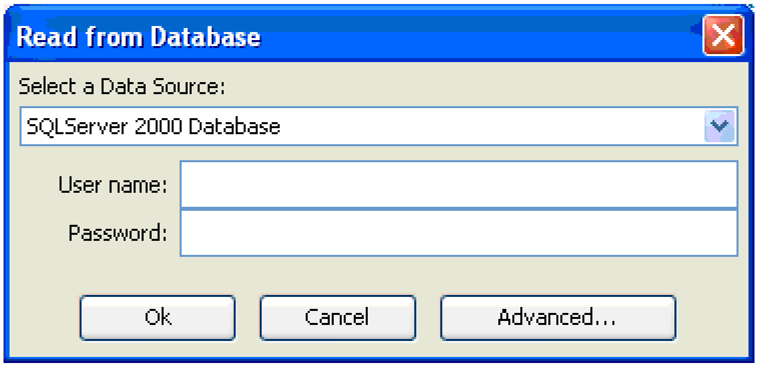
Adding custom database connectivity

In a production environment, you may want to avoid having to enter URL and credential information at every database read or save operation. You can declare *data source* extension points in a plug-in to define all parameters needed to access a given database instance.

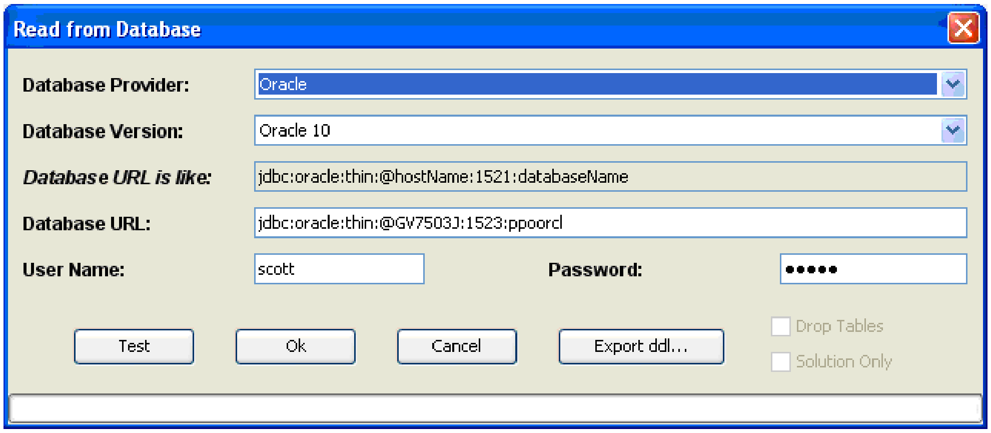
In the “`plugin.xml`” file you can define multiple xml elements as follows:

```
<extension point="DBDataSource" id="databse1" name="Oracle 10 Database"
    database="Oracle10"
    url="jdbc:oracle:thin:@GV7503J:1523:ppoorcl"
    userName="scott"
    password="tiger"
    hideCredentials="true">
</extension>
<extension point="DBDataSource" id="databse2" name="SQLServer 2000 Database"
    database="SQLServer2000"
    url="jdbc:sqlserver://GV7503J:1433"
    saveSolutionOnly="true">
</extension>
```

After deploying the plug-in, when the user selects the menu item **File -> Open Scenario from Database** or **File -> Save Scenario in Database**, the **Read from Database** dialog box displays requesting the data source, and user and password information if not already provided in the plug-in file.



The **Advanced** button displays the generic database access dialog box.


Read from Database

Database Provider: Oracle

Database Version: Oracle 10

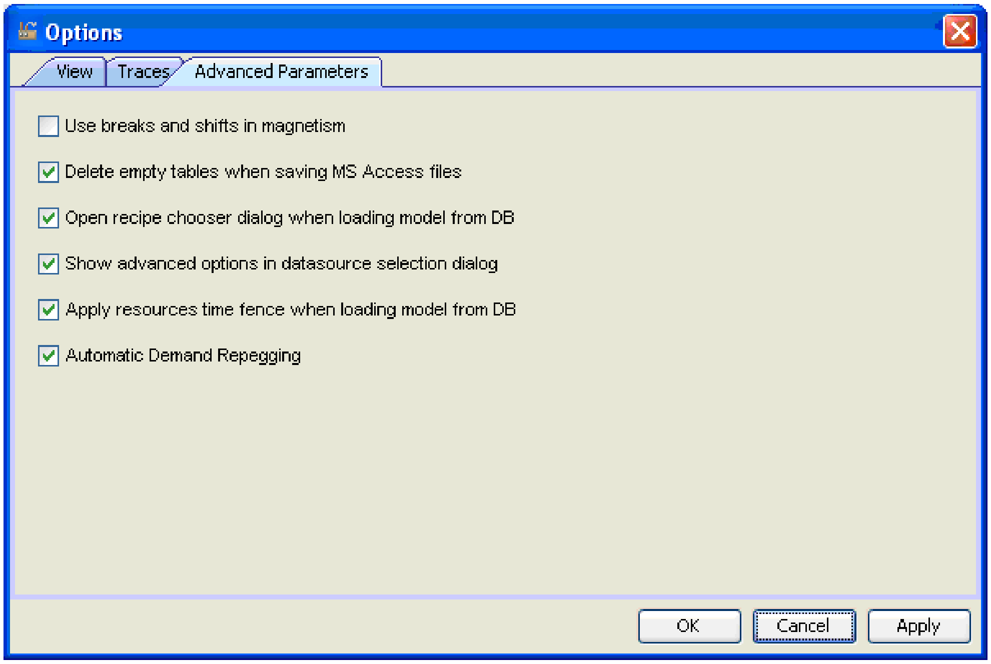
Database URL is like: jdbc:oracle:thin:@hostName:1521:databaseName

Database URL: jdbc:oracle:thin:@GV7503J:1523:ppoorcl

User Name: scott

Password: ●●●●●

The advanced options button can be hidden with the menu item **Tools —> Options —> Advanced Parameters**.


Options

View | Traces | Advanced Parameters

- [ ] Use breaks and shifts in magnetism
- [✓] Delete empty tables when saving MS Access files
- [✓] Open recipe chooser dialog when loading model from DB
- [✓] Show advanced options in datasource selection dialog
- [✓] Apply resources time fence when loading model from DB
- [✓] Automatic Demand Repegging

Using service levels, lead time, and demand variability to manage stock levels

This section describes how to manage stock inventory levels to help prevent stock outages even when faced with uncertain demand and production time.

In this section

Manually defining stock levels and corridors
Using stock cost functions or "days of supply" targets is easy to implement but has limitations.

Service level concepts
Describes managing stock levels through the use of service levels, including the capacity to handle forecast error and variability in demand and production lead time.

Examples of using service levels, demand variability, and production lead time variability
This section includes examples that show the effects of using service level functionality. The examples use the file `yogurt_factory.csv` which is available from the GUI start page and in the standard distribution. The examples are best followed sequentially.

Manually defining stock levels and corridors

One way to ensure that you have adequate levels of stock is to define inventory cost functions or apply a minimum and maximum “days of supply” requirement to compute inventory levels. These methods are relatively easy to apply, but do have some limitations.

Demand for products, even in the short term, may vary considerably due to many uncontrollable factors, leading to forecasting errors. Production lead time must be assumed yet may also vary. The planner must “manually” include these concerns into a cost function or days of supply target, leading to the possibility of not having sufficient inventory or of maintaining too much stock at an unnecessary cost. Inevitably, these methods rely heavily on the forecasting abilities of the planner.

A more precise method of managing stock levels is to implement industry-standard *service levels*, and take into account variability in demand and production lead time as part of the planning model. The following sections describe how to utilize these technologies in your PPO data model.

Service level concepts

Describes managing stock levels through the use of service levels, including the capacity to handle forecast error and variability in demand and production lead time.

In this section

Uncertain demand and forecast error
How PPO lets you manage demand uncertainty.

Production lead time
How PPO lets you manage production lead time.

Service levels
Implementing service levels in your data model.

Uncertain demand and forecast error

Demand for products may vary, and forecasting efforts may not always be accurate. PPO calls this uncertainty *demand variability*, and you can model it based on the following assumptions.

♦ Demand variability depends only on the demanded material.

♦ Demand variability is assumed to follow normal distribution laws, with an average value read from the data. The standard deviation is computed using a coefficient of demand variation.

♦ Demand variability is taken into account for a given planning bucket, over a period extending from the bucket end up to an average lead time, which depends only on the material.

The DEMAND_VARIABILITY column in the PPO_MATERIAL table allows you to model demand variability. This field is a ratio with a value between 0 and 1, and is used to compute the deviation of a demand for this material. Its default is zero (fully certain).

Demand variability data can also be provided in the API. In the GUI, **Demand Variability** is available in the **Service Level** area (see *Service level in the GUI*).

Production lead time

PPO allows you to include production lead time data with the service level model by following these assumptions:

- ♦ The lead time follows a normal probability distribution, having the expected value of the average lead time. The deviation of lead time is represented by the lead time standard deviation, which is modeled as a duration expressed in time units with a default of zero.
- ♦ When the lead time standard deviation is not null, the computation of the target stock takes into account an additional variance contribution. This contribution estimates the potential extra demand that may occur during an extra period of lead time.

All service level types can incorporate lead time uncertainty.

The PPO_MATERIAL table models the average lead time in the AVERAGE_LEAD_TIME field, and variation is provided in the LEAD_TIME_STD_DEVIATION field as a duration with a default of zero.

Lead time data can also be provided in the API. In the GUI, lead time data is available in the **Service Level** area (see *Service level in the GUI*).

Service levels

Implementing service levels in your data model.

In this section

Service level types and targets
Describes the theory and definitions you need to know to implement service levels.

Model elements for service level
Describes the schema elements to model the service level.

Computing target stock from the service level
Describes the computation of the inventory target stock at the end of each time bucket to ensure the desired service level.

Service level target stock in planning
The planning engine uses the service level stock values as targets in its objective function.

Service level types and targets

A service level essentially denotes a KPI to measure customer satisfaction. For a given material you provide a service level *type* (for example *fill rate*) and a service level *target* (a numeric value) which together define your desired service level.

There are two primary service level types: **Service Level** (also known as *cycle*), and **Fill Rate**. Both of these also have *dynamic* versions. The service level target can be between 50% and 100% (not allowed).

Along with the service level type and the service level target value, PPO includes demand variability (with forecasting error) and production lead time values to compute the stock level necessary to reach your defined service level. This calculated stock level represents an ideal value that the PPO engine will try to achieve as closely as possible; we refer to it as the *target stock level*, *service level target stock*, or simply, *target stock*. When you have implemented service levels, the *target stock* is represented in the GUI by the stock min (red) line on the **Stock Coverage** view.

Cycle service level

The *Cycle Service Level* measures the probability that a *stock out* event will not happen. A stock out event occurs in a given time bucket if the total demand due in the bucket exceeds the inventory quantity that is available in the bucket. The cycle service level sets a limit on the frequency of stock out events. The service level target is between 50% and 100% (not allowed) and denotes the minimal frequency of non-stock-out events. For example, a service level target value of 95% means that, on average, no more than five cycles (buckets) out of 100 will contain stock out events. The cycle service level is known variously as the *alpha*, *type 1*, or *event-based service level*. In the PPO GUI it is usually referred to simply as **Service Level**.

Fill rate service level

The *Fill Rate Service Level* measures the expected percentage of *overall demand quantity* which is satisfied. It denotes the minimum demand satisfaction ratio that can be expected on average (while dealing with uncertain demands following probability distributions). This level is based on the overall quantity of customer demand that is met by stock on hand, irrespective of the number of cycles (buckets) when stock out events occur. The associated service level target is a value between 50% and 100% (not allowed) representing the ratio of satisfied demand. For example, a target value of 98% means that the expected demand satisfaction ratio based on quantity must be over 98%. This level is referred to as **Fill Rate** in the PPO GUI but is also known as the *beta*, *type 2*, or *quantity-based service level*.

Dynamic service levels

PPO uses service level data to compute a target stock level based on dependent and independent demands, using stochastic computations. The target stock level is computed by estimating the variance of the demand over a fixed look-ahead period, using an approximation of when the next production will occur.

Dynamic service levels take this process further by exactly taking into account the time of the next production. For each bucket, a target stock level is computed using the exact demand variance until the next production time.

Consider an example using daily buckets with the next production five days away, and an average lead time of two days. The target stock level is computed from the cumulated demand over the next five days to cover the risk until the next production.

Model elements for service level

The service level data consists of the *type* and the *target* value, and is modeled using two columns in the PPO_MATERIAL table:

- SERVICE_LEVEL_TYPE: An enumerated value which describes the type of service level used to compute the target stock. Possible values are: `Disabled`, `ServiceLevel`, `FillRate`, `ServiceLevelDynamic`, and `FillRateDynamic`. The default value `Disabled` means that no service level computation is used, so the stock values are computed either through the days of supply or by an explicit inventory min function.
- TARGET_SERVICE_LEVEL: A float value between 0.5 and 1 (not allowed), used to denote the target value of the service level. The semantics of this target value depend on which type of service level is active for the material. The default value is 0.95.

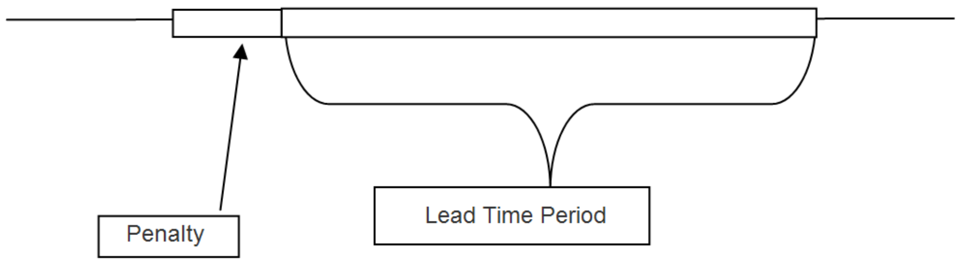
Note that service level type and target value are properties of the material, and are independent of time.

Computing target stock from the service level

Once an active service level type has been assigned to a material, PPO computes a target stock level from the service level information. To accomplish this feat PPO estimates for each bucket the variance of the demand in the future of the bucket.

This period extends from the bucket’s end time with a duration equal to the material’s average lead time for static types; for dynamic types, this duration extends to the next production. PPO computes the total demand variance over this look-ahead period from the bucket, and computes a target stock from this variance. The precise way to compute the stock depends on the service level type and involves stochastic computations based on the normal distribution.

When active, the target stock computation supersedes the usual computation of minimum stock, even if a minimum days of supply (or explicit minimum inventory function) has been provided. It does not impact the computation of a maximum stock, if a maximum days of supply value has been provided.

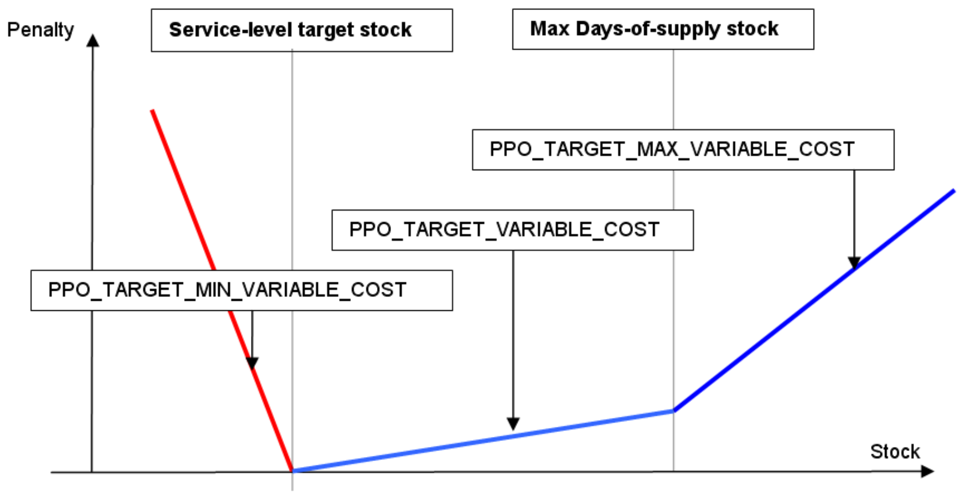


Service level target stock in planning

As described previously, PPO computes a target stock level from the service level data, which supersedes the minimum days of supply. The target stock is computed to guarantee a minimal service level value, guarding against random variations of demands during a look-ahead period (equal to the average lead time). This target stock level is the sum of the total demand quantity in the look-ahead period (starting at the bucket's end with average duration lead time, and of a safety stock quantity calculated from the demand variability).

The calculated target stock is displayed in the PPO GUI as the minimum stock curve (red line) in the **Stock Coverage** view. The maximum stock curve is not impacted by the service level, and is still calculated from the maximum days of supply. As the target stock is actually an ideal stock level, a maximum days of supply stock value is not necessary when the service level is active, although it can be used to enforce penalties.

Costs are calculated by the planning engine when actual stock levels deviate from the target stock level. Deviations below the target stock level are penalized using the PPO_MATERIAL|TARGET_MIN_VARIABLE_COST value, and deviations above are penalized using PPO_MATERIAL|TARGET_VARIABLE_COST (if this cost is zero, deviations above the target stock are not penalized). The diagram below shows how PPO computes costs for deviations from the service level stock value.


Penalty
Service-level target stock
Max Days-of-supply stock
PPO_TARGET_MAX_VARIABLE_COST
PPO_TARGET_VARIABLE_COST
PPO_TARGET_MIN_VARIABLE_COST
Stock

Examples of using service levels, demand variability, and production lead time variability

This section includes examples that show the effects of using service level functionality. The examples use the file `yogurt_factory.csv` which is available from the GUI start page and in the standard distribution. The examples are best followed sequentially.

In this section

Service level in the GUI
Describes how to access service level data and stock results in the PPO GUI..

Using the cycle service level
An example of editing the cycle service level data.

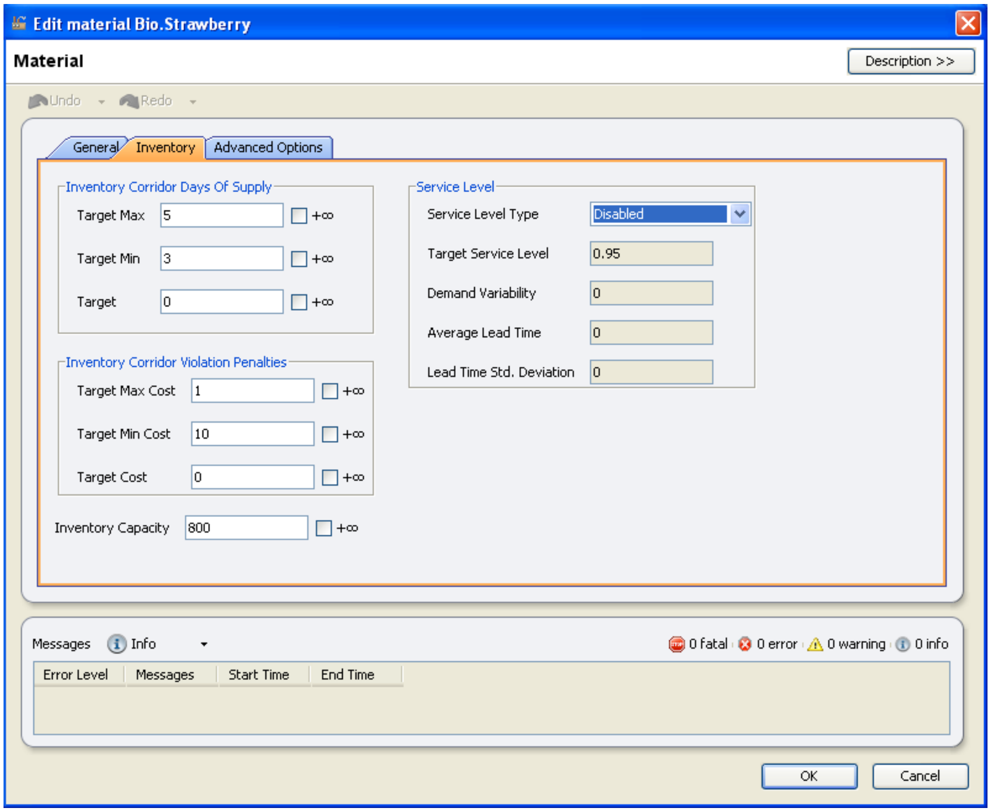
Using the fill rate service level
An example of editing the fill rate service level data.

Dynamic service level example
Dynamic service level takes into account the time of the next production.

Uncertain production lead time example
Describes how to model uncertain lead time in PPO.

Service level in the GUI

The service level data is available for viewing and editing through the **Materials** tab of the **Master Data** view. Navigate to that view, then right-click on a material to display the contextual menu. Select **Edit**, and the material data editing window displays. Select the **Inventory** tab and a window similar to the following image displays.



Inventory Corridor Days Of Supply

Target Max	5	☐ +∞
Target Min	3	☐ +∞
Target	0	☐ +∞

Inventory Corridor Violation Penalties

Target Max Cost	1	☐ +∞
Target Min Cost	10	☐ +∞
Target Cost	0	☐ +∞

Inventory Capacity 800 ☐ +∞

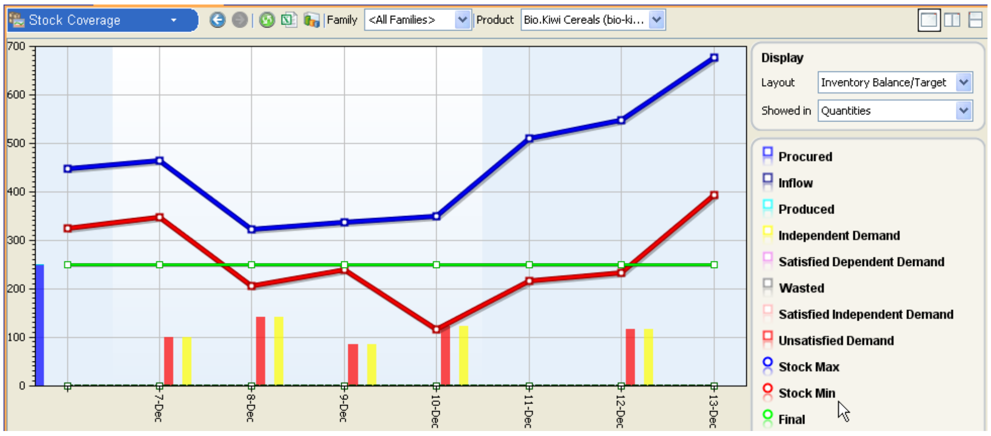
Service Level

Service Level Type	Disabled
Target Service Level	0.95
Demand Variability	0
Average Lead Time	0
Lead Time Std. Deviation	0

Error Level	Messages	Start Time	End Time

In this window you have access to all the inventory management controls: Service levels, days of supply targets, and inventory costs. Within the **Service Level** area, you can set the service level type, target, demand variability, the average lead time and deviation. The examples that follow refer to this **Inventory** tab to manipulate data to show the different effects of using service level technology.

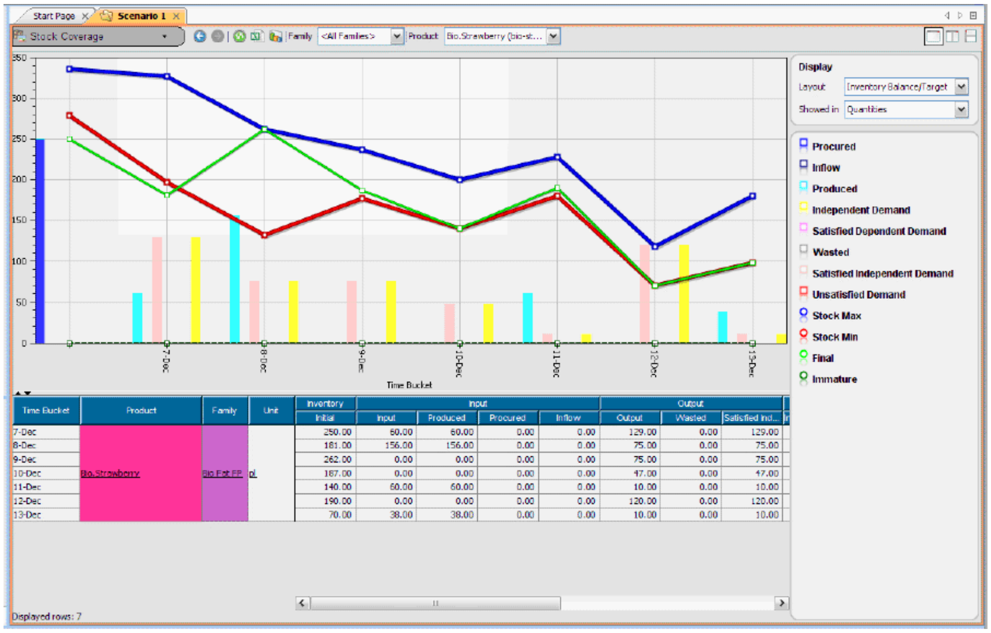
The results of using inventory controls can be seen in the **Stock Coverage** view. A partial image of this view follows. The red **Stock Min** line is of particular interest, because when a service level has been enabled, this line represents the ideal service level target stock that is calculated. When service levels are disabled, this line represents the stock min calculated from the appropriate inventory cost function or days of supply target.



When you edit the service level data, the stock curve on this view is updated. However, the current solution data cannot be updated until you relaunch a solve. So after editing the service level information, optimize the scenario again to ensure that the displayed planning solution is consistent with the master data.

Using the cycle service level

The following image shows the example file `yogurt_factory.csv` loaded in PPO, optimized, and open to the **Stock Coverage** view. The stock corridor for the **Bio.Strawberry** material is displayed according to days of supply, between 3 (min) and 5 (max) days of supply. (Images shown here may not exactly match results displayed in your version of PPO for a variety of reasons.)



Layout: Inventory Balance/Target

Showed in: Quantities

Time Bucket	Product	Family	Unit	Inventory	Input				Output		
				Initial	Input	Produced	Procured	Inflow	Output	Wasted	Satisfied Ind...
7-Dec				250.00	60.00	60.00	0.00	0.00	129.00	0.00	129.00
8-Dec				181.00	156.00	156.00	0.00	0.00	75.00	0.00	75.00
9-Dec				262.00	0.00	0.00	0.00	0.00	75.00	0.00	75.00
10-Dec	Bio.Strawberry	Bio Fat FP	pl	187.00	0.00	0.00	0.00	0.00	47.00	0.00	47.00
11-Dec				140.00	60.00	60.00	0.00	0.00	10.00	0.00	10.00
12-Dec				190.00	0.00	0.00	0.00	0.00	120.00	0.00	120.00
13-Dec				70.00	38.00	38.00	0.00	0.00	10.00	0.00	10.00

Displayed rows: 7

We'll introduce service level into this example, by editing the material inventory data. (Accessing the **Inventory** editing window is described in *Service level in the GUI*).

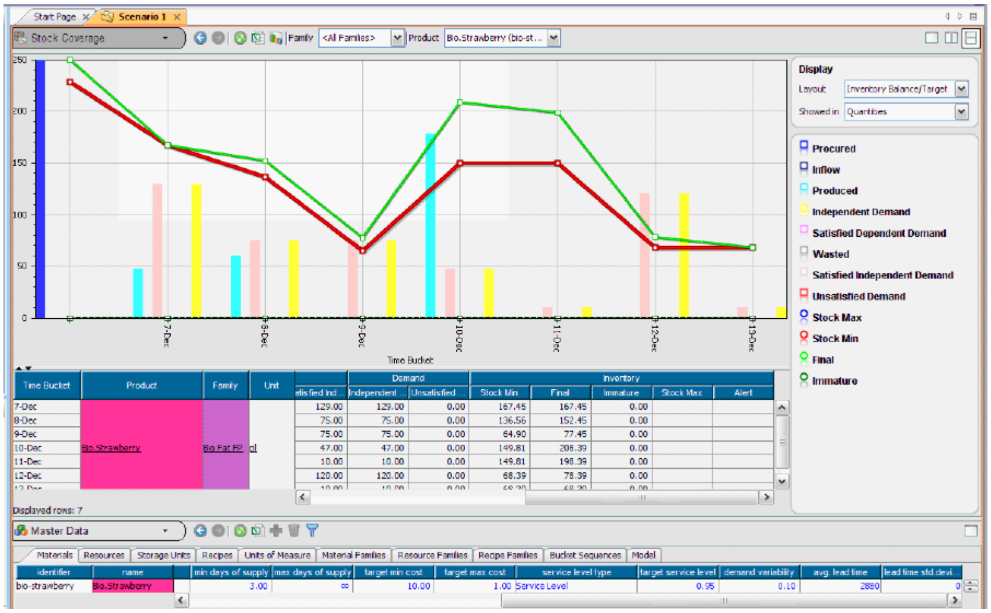
Split the PPO view into two views, with the **Master Data** view in the bottom pane as shown in the image below. Edit the Bio.Strawberry material, and on the **Inventory** tab, set the **Service Level Type** field to **Service Level**, and select **OK**. Notice that in the **Stock Coverage** view, the maximum stock curve does not change for Bio.Strawberry, but the stock min curve (now acting as the service level target stock curve) goes to zero. This happens because the average lead time is zero, and both total demand in the look-ahead period and demand variability are zero.

Edit the material again. Note that the **Inventory Capacity** is set to **800**. Change the maximum days of supply to positive infinity (**Target Max** of **Inventory Corridor Days of Supply**), and select **OK**. This action removes it from view so that we can focus on the target and final stock curves.

Edit the material again, changing the **Average Lead Time** to two days (2880, assuming time units in minutes). The stock min (target stock) curve changes again reflecting the fact that for each bucket, stock min is now equal to two days of demand. There is no variability of demand however, so edit the material and change **Demand Variability** to 0.1 (10%).

Select **OK** and the target stock curve (stock min curve) increases lightly as it now takes into account demand variability.

Examine the values shown in the **Stock Min** column of the **Stock Coverage** view, as shown in this image representing all the changes made thus far.



Time Bucket	Product	Family	Unit	Demand			Inventory				
				atisfied Ind...	Independent ...	Unsatisfied ...	Stock Min	Final	Immature	Stock Max	Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	167.45	167.45	0.00		
8-Dec				75.00	75.00	0.00	136.56	152.45	0.00		
9-Dec				75.00	75.00	0.00	64.90	77.45	0.00		
10-Dec				47.00	47.00	0.00	149.81	208.39	0.00		
11-Dec				10.00	10.00	0.00	149.81	198.39	0.00		
12-Dec				120.00	120.00	0.00	68.39	78.39	0.00		

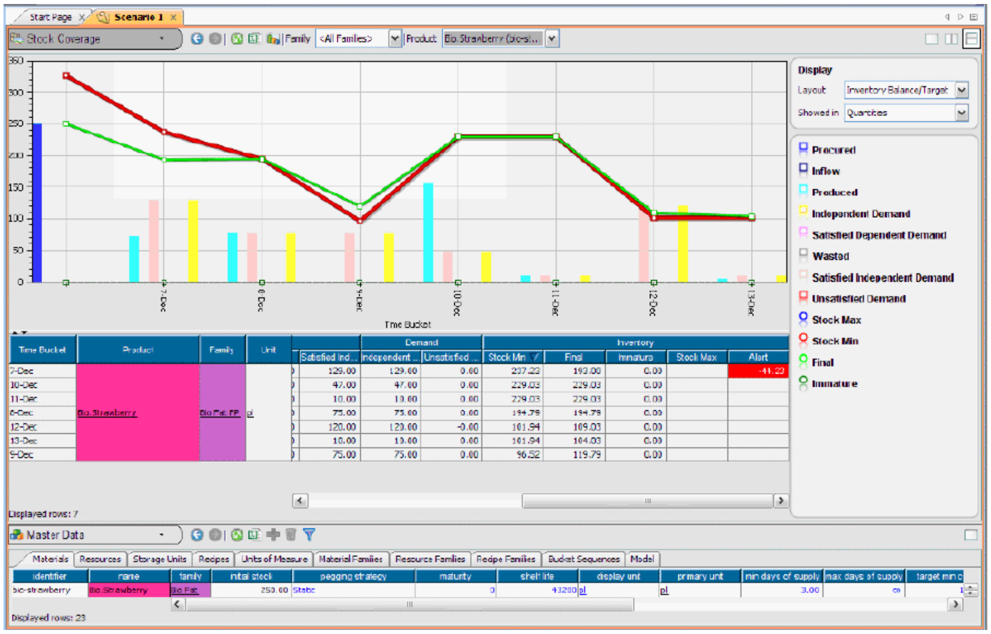
Displayed rows: 7

identifier	name		min days of supply	max days of supply	target min cost	target max cost	service level type	target service level	demand variability	avg. lead time	lead time std.devi...
bio-strawberry	Bio.Strawberry		3.00	∞	10.00	1.00	Service Level	0.95	0.10	2880	0

The effect of using this 95% service level, low demand variability, and short average lead time is to lower (overall) the minimum stock levels as compared to the default case of this example file, which uses the three “days of supply” data. Be sure to compare values after optimization.

After an optimization with the changes made so far, the final stock curve is above the target stock, because deviations below the target stock are penalized using the **Target Min Cost** value (equal to 10). Deviations above the target cost incur no cost because the **Target Cost** is zero by default. If you edit this material again and set the target cost to 1.0 and relaunch optimization, in the resulting plan you’ll see that the final stock curve changes. Costs are now incurred for deviations above or below the target level, so optimization has an incentive to keep the final stock curve closer to the target stock level (stock min).

Next, increase the demand variability to 0.5 (50%) and relaunch optimization. This results in the following image, where the minimum stock curve increases significantly. As demand variability increases, so does the variance of the demand in the lead-time period; this causes target stock levels to increase in all buckets.



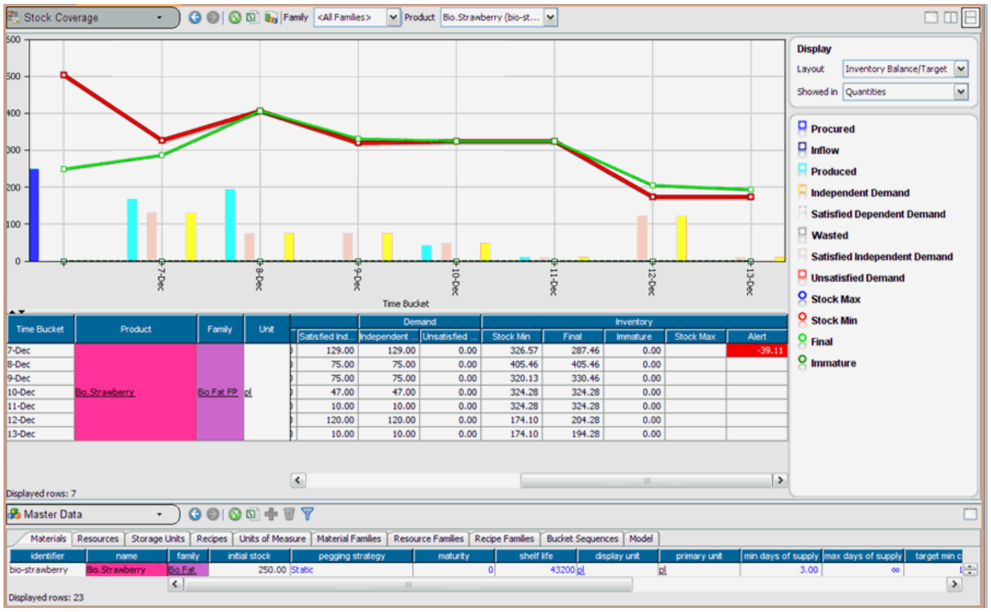
Time Bucket	Product	Family	Unit	Demand: Satisfied Ind...	Demand: Independent ...	Demand: Unsatisfied ...	Inventory: Stock Min	Inventory: Final	Inventory: Immature	Inventory: Stock Max	Inventory: Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	237.23	193.00	0.00		-44.23
10-Dec				47.00	47.00	0.00	229.03	229.03	0.00		
11-Dec				10.00	10.00	0.00	229.03	229.03	0.00		
8-Dec				75.00	75.00	0.00	194.79	194.79	0.00		
12-Dec				120.00	120.00	-0.00	101.94	109.03	0.00		
13-Dec				10.00	10.00	0.00	101.94	104.03	0.00		
9-Dec				75.00	75.00	0.00	96.52	119.79	0.00		

Displayed rows: 7

identifier	name	family	initial stock	pegging strategy	maturity	shelf life	display unit	primary unit	min days of supply	max days of supply	target min c
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	0	43200	pl	pl	3.00	∞	1

Displayed rows: 23

Next, increase the average lead time to four days (5760) and the target service level to 0.98, and relaunch an optimization. The target stock level (stock min curve) increases significantly.



Time Bucket	Product	Family	Unit	Demand: Satisfied Ind...	Demand: Independent ...	Demand: Unsatisfied ...	Inventory: Stock Min	Inventory: Final	Inventory: Immature	Inventory: Stock Max	Inventory: Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	326.57	287.46	0.00		-39.11
8-Dec				75.00	75.00	0.00	405.46	405.46	0.00		
9-Dec				75.00	75.00	0.00	320.13	330.46	0.00		
10-Dec				47.00	47.00	0.00	324.28	324.28	0.00		
11-Dec				10.00	10.00	0.00	324.28	324.28	0.00		
12-Dec				120.00	120.00	0.00	174.10	204.28	0.00		
13-Dec				10.00	10.00	0.00	174.10	194.28	0.00		

Displayed rows: 7

identifier	name	family	initial stock	pegging strategy	maturity	shelf life	display unit	primary unit	min days of supply	max days of supply	target min c
bio-strawberry	Bio.Strawberry	Bio Fat	250.00	Static	0	43200	pl	pl	3.00	∞	1

Displayed rows: 23

To summarize, the cycle service level causes the target stock level to change as follows:

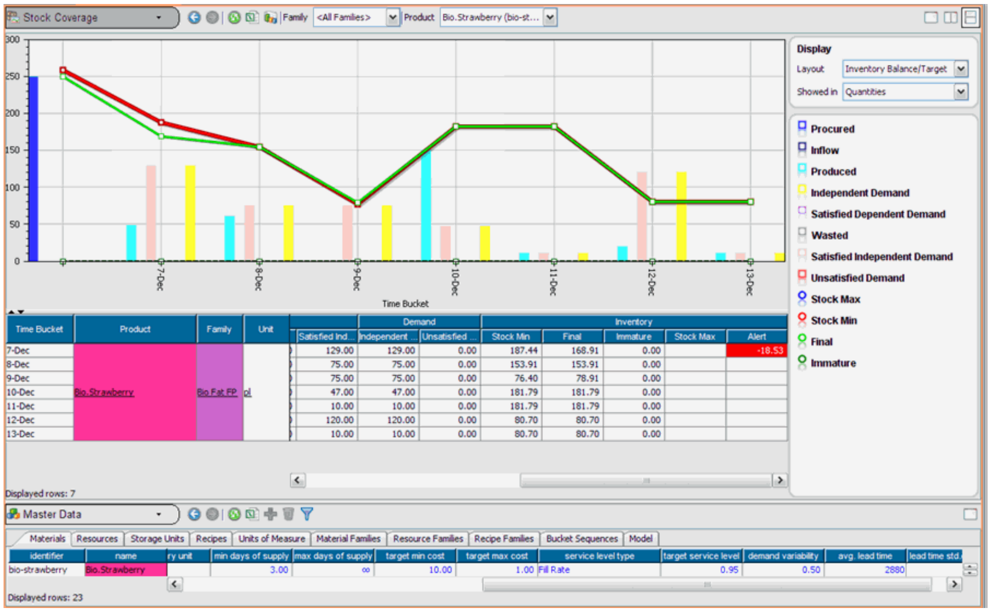
- Increase with demand variability
- Increase with the average lead time
- Increase (slowly) with the target service level.

Using the fill rate service level

The previous example showed the effects of using the cycle service level; this example takes the same problem and applies the fill rate service level. All else being equal, it is intuitive that meeting a percentage of total demand quantity should be less constraining than ensuring the same percentage of non-stock-out events. So we can presume that switching the service level type to **Fill Rate** should lower the computed target stock levels, as compared to using the cycle service level.

Using the example file `yogurt_factory.csv`, edit the Bio.Strawberry material, and on the **Inventory** tab change the **Service level Type** to **Fill Rate**. (Accessing the **Inventory** editing window is described in *Service level in the GUI*). Some of the other settings include a service level target of 0.98, average lead time of 5760, variability of 0.5, and a target cost of 1.

Optimize the scenario, and notice that the target stock (stock min) values are lower than with the cycle service level.



Time Bucket	Product	Family	Unit	Satisfied Ind...	Demand: Independent ...	Demand: Unsatisfied ...	Inventory: Stock Min	Inventory: Final	Inventory: Immature	Inventory: Stock Max	Inventory: Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	187.44	168.91	0.00		-18.53
8-Dec				75.00	75.00	0.00	153.91	153.91	0.00		
9-Dec				75.00	75.00	0.00	76.40	78.91	0.00		
10-Dec				47.00	47.00	0.00	181.79	181.79	0.00		
11-Dec				10.00	10.00	0.00	181.79	181.79	0.00		
12-Dec				120.00	120.00	0.00	80.70	80.70	0.00		
13-Dec				10.00	10.00	0.00	80.70	80.70	0.00		

Displayed rows: 7

identifier	name	ry unit	min days of supply	max days of supply	target min cost	target max cost	service level type	target service level	demand variability	avg. lead time	lead time std.
bio-strawberry	Bio.Strawberry		3.00	∞	10.00	1.00	Fill Rate	0.95	0.50	2880	

Displayed rows: 23

To summarize, the fill rate target stock level varies as follows:

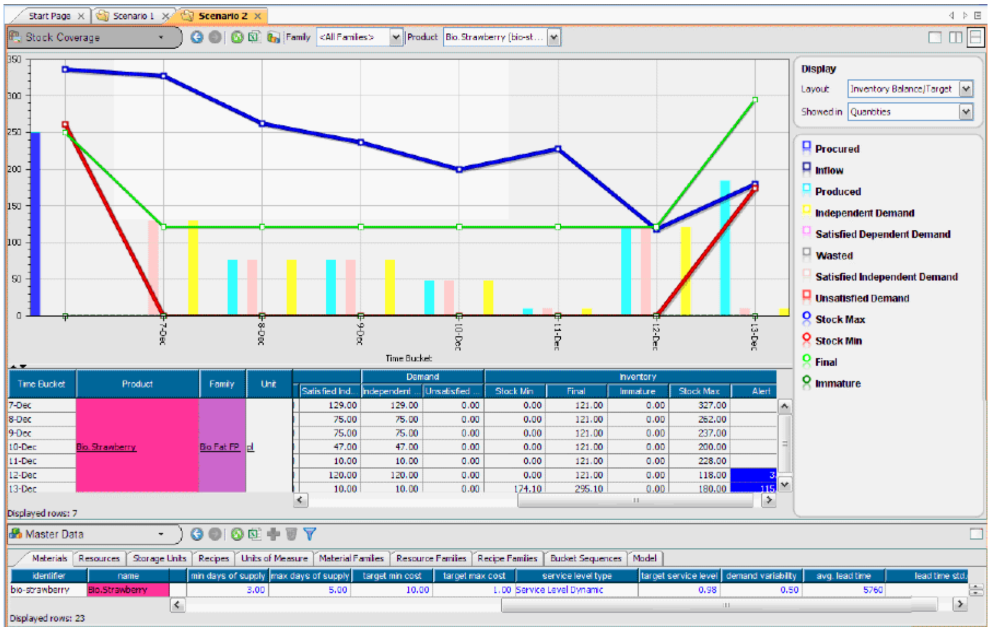
- Increases with demand variability
- Increases with the lead time
- Increases (slowly) with the target service level.

Dynamic service level example

Dynamic service levels incorporate the time of the next production when calculating target stock levels.

Use the example file `yogurt_factory.csv` and edit the Bio.Strawberry material. On the **Inventory** tab, set the type to **Service Level**, with 50% (0.5) demand variability, 98% target service level and a four day (5760) average lead time. (Accessing the **Inventory** editing window is described in *Service level in the GUI*)

Optimize and check the results. Then edit the material again, this time selecting **Service Level Dynamic**, and optimize. The results should be similar to the following image. In the displayed corridor, all stock min values are zero for buckets that precede a production. So once again, the look-ahead period is empty, hence both demand variance and lead-time expected demand are null.



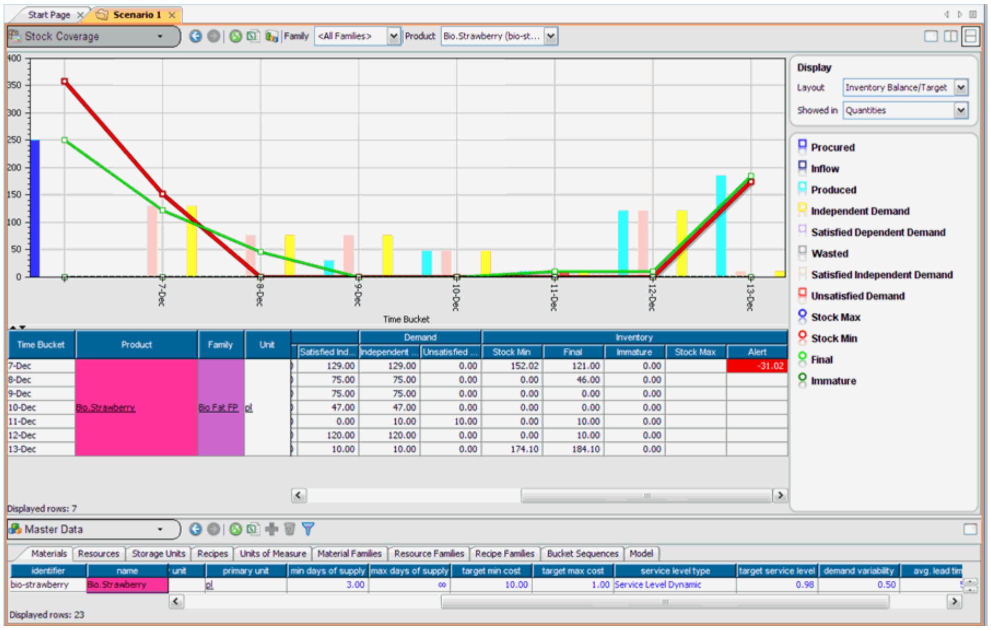
Time Bucket	Product	Family	Unit	Satisfied Ind... (Demand)	Independent ... (Demand)	Unsatisfied ... (Demand)	Stock Min (Inventory)	Final (Inventory)	Immature (Inventory)	Stock Max (Inventory)	Alert (Inventory)
7-Dec				129.00	129.00	0.00	0.00	121.00	0.00	327.00	
8-Dec				75.00	75.00	0.00	0.00	121.00	0.00	262.00	
9-Dec				75.00	75.00	0.00	0.00	121.00	0.00	237.00	
10-Dec	Bio.Strawberry	Bio Fat FP	pl	47.00	47.00	0.00	0.00	121.00	0.00	200.00	
11-Dec				10.00	10.00	0.00	0.00	121.00	0.00	228.00	
12-Dec				120.00	120.00	0.00	0.00	121.00	0.00	118.00	3
13-Dec				10.00	10.00	0.00	174.10	295.10	0.00	180.00	115

Displayed rows: 7

identifier	name		min days of supply	max days of supply	target min cost	target max cost	service level type	target service level	demand variability	avg. lead time	lead time std.
bio-strawberry	Bio.Strawberry		3.00	5.00	10.00	1.00	Service Level Dynamic	0.98	0.50	5760	

Displayed rows: 23

Next, modify the planned production and examine how this affects the stock levels. Go to the **Master Data** view, **Recipes** tab, and for the **bio-strawberry** recipe, edit the **valid start** column to show a date of "9 Dec 2006 00:00:00" instead of minus infinity. This will prevent material production on December 8th, as occurs in the previous image. Launch planning again and look at the **Stock Coverage** view.



Time Bucket	Product	Family	Unit	Demand: Satisfied Ind...	Demand: Independent ...	Demand: Unsatisfied ...	Inventory: Stock Min	Inventory: Final	Inventory: Immature	Inventory: Stock Max	Inventory: Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	152.02	121.00	0.00		-31.02
8-Dec				75.00	75.00	0.00	0.00	46.00	0.00		
9-Dec				75.00	75.00	0.00	0.00	0.00	0.00		
10-Dec				47.00	47.00	0.00	0.00	0.00	0.00		
11-Dec				0.00	10.00	10.00	0.00	10.00	0.00		
12-Dec				120.00	120.00	0.00	0.00	10.00	0.00		
13-Dec				10.00	10.00	0.00	174.10	184.10	0.00		

Displayed rows: 7

identifier	name	unit	primary unit	min days of supply	max days of supply	target min cost	target max cost	service level type	target service level	demand variability	avg. lead tim
bio-strawberry	Bio.Strawberry		pl	3.00	∞	10.00	1.00	Service Level Dynamic	0.98	0.50	

Displayed rows: 23

The first production of Bio.Strawberry has shifted to Dec 9th, and the target stock level has been recomputed accordingly.

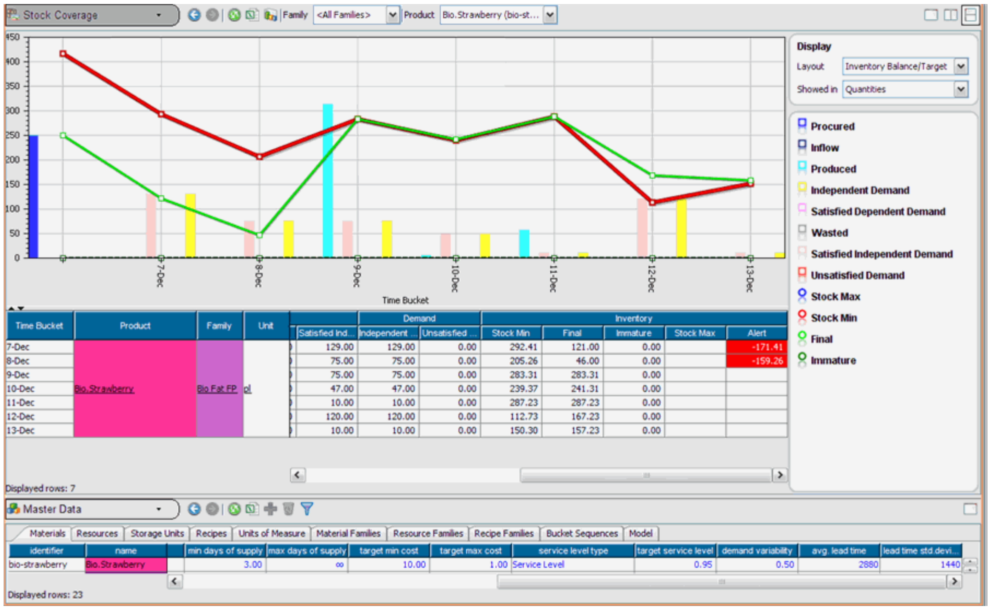
To summarize dynamic service levels:

- ♦ Dynamic service levels estimate demand variability up to the exact next production time, not using the approximation of average lead time.
- ♦ Alpha and beta (cycle and fill rate service level) dynamic types behave in the same way; only the method of taking into account demand variability varies.

Uncertain production lead time example

To demonstrate how uncertain lead time affects stock calculations, use the example file `yogurt_factory.csv` and edit the Bio.strawberry material. On the **Inventory** tab, set the type to **Service Level**, with a 95% target level, and 50% (0.50) demand variability. Let's assume that the production lead time is two days, plus or minus one day. So set **Average Lead Time** to 2880 and **Lead Time Std. Deviation** to 1440.

Optimize the scenario and check the **Stock Coverage** view.



Time Bucket	Product	Family	Unit	Demand			Inventory				
				Satisfied Ind...	Independent ...	Unsatisfied ...	Stock Min	Final	Immature	Stock Max	Alert
7-Dec	Bio.Strawberry	Bio Fat FP	pl	129.00	129.00	0.00	292.41	121.00	0.00		-171.41
8-Dec				75.00	75.00	0.00	205.26	46.00	0.00		-159.26
9-Dec				75.00	75.00	0.00	283.31	283.31	0.00		
10-Dec				47.00	47.00	0.00	239.37	241.31	0.00		
11-Dec				10.00	10.00	0.00	287.23	287.23	0.00		
12-Dec				120.00	120.00	0.00	112.73	167.23	0.00		
13-Dec				10.00	10.00	0.00	150.30	157.23	0.00		

Displayed rows: 7

identifier	name		min days of supply	max days of supply	target min cost	target max cost	service level type	target service level	demand variability	avg. lead time	lead time std.devi...
bio-strawberry	Bio.Strawberry		3.00	∞	10.00	1.00	Service Level	0.95	0.50	2880	1440

Displayed rows: 23

Lead time uncertainty introduces an additional factor in the accumulated demand variance for each bucket; therefore minimum stock increases with lead time deviation.

Decomposition framework

How to decompose a planning problem into smaller components.

In this section

Introduction to decomposition
Explains why decomposition is useful and how to do it.

Building the decomposition framework
Using database tables to perform the problem decomposition on the yogurt manufacturing process.

Optimization using scopes in the GUI
The result seen in the GUI.

Advanced usage of the decomposition framework using Java API
Using a plug-in and API to decompose the problem.

Introduction to decomposition

Creating a decomposition framework is useful for two main reasons. One reason is to allow planners to separate the problem or process so that they can validate one level before tackling the next part of the process. Another reason is simply that the problem is too difficult or time consuming to be appropriately solved, and so decomposition is used to reduce the problem complexity.

The decomposition process

This section describes some general ideas about how to implement a decomposition.

First, planners are usually familiar with decomposing the problem at their plant, and that experience is valuable. Following a known and tested decomposition strategy may prevent duplication of effort, thereby saving time and analysis work. This approach also allows planners to gain better insight into the problem solving methods of PPO, since they already understand the decomposition itself.

Tracking and understanding the material flow is important. In most manufacturing cases a semi-finished material or product is prepared, which is then transformed into a finished product. These two processes (producing the semi-finished material and producing the finished product) can therefore be solved separately and consecutively providing a natural break point for decomposition.

Additionally, a crucial principle based on constraint theory is to **identify the bottleneck point or level and solve this stage of the process first**.

These factors are used to help decide between two types of decomposition: *Pull by demand*, or *Push raw materials*.

Pull by demand

Pull by demand means to solve the finished products first and freeze the associated planning schedule. Then, enough semi-finished product must be produced in time to feed the finished product production lines.

The advantage of this approach is that the finished product orders are in phase with real demand. A potential disadvantage is that you may not be able to produce all the semi-finished material needed to supply the filling and finishing lines as scheduled. The result is that it may not be possible to supply all the finished product to meet demand.

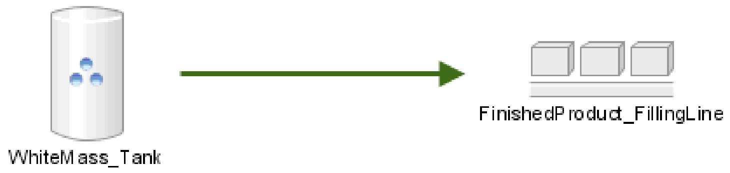
Push raw materials

The push raw materials approach means to solve the semi-finished production first, and freeze the associated planning schedule. Then the semi-finished production is used to input materials for the finished product calculation.

The advantage of this approach is that raw materials are efficiently utilized to produce the semi-finished product. A potential disadvantage is that you could produce more semi-finished material than is necessary to supply the finished product lines, resulting in possible waste.

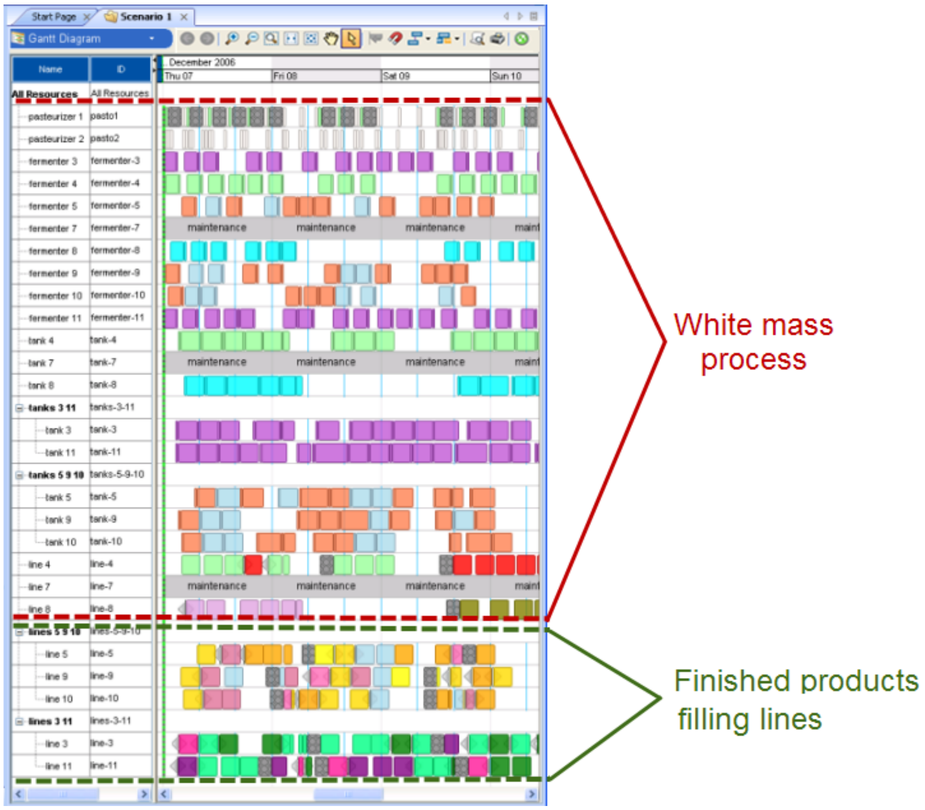
Building the decomposition framework

This section demonstrates the creation of a decomposition framework in the context of the yogurt manufacturing example. In the dairy industry, the semi-finished material or product is called the *white mass*. So the first part of making fresh yogurt is to create the white mass with pasteurizers, fermenters, and storage tanks. The white mass is then used in the filling and finishing lines to create the finished product, fresh yogurt.



The aim of this decomposition example is to first build the schedule of finished products. Given that schedule, next create a schedule of semi-finished white mass production that provides the materials necessary to support the finished products schedule. This is pull by demand decomposition.

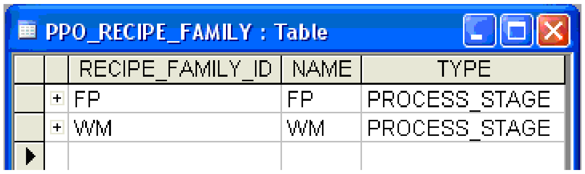
The result is displayed in the following **Gantt Diagram**, with the finished product schedule outlined in green and the white mass process outlined in red.



The general principle is to group recipes by similar level in the production process in order to describe the decomposition. Then we can describe the solving approach of this decomposition by associating a scope with the optimization profile.

Each recipe must be attached to a recipe family. The recipe family is used to describe a stage of the production process, and so group recipes appropriately for the decomposition.

We start modeling the decomposition with the PPO_RECIPE_FAMILY table used to create two types of recipe families. Note that the **TYPE** column below contains the same value for both the finished product and white mass recipes; this is to indicate that the two recipe families are members of the same partition layer of the problem; in this example, that is called the PROCESS_STAGE.



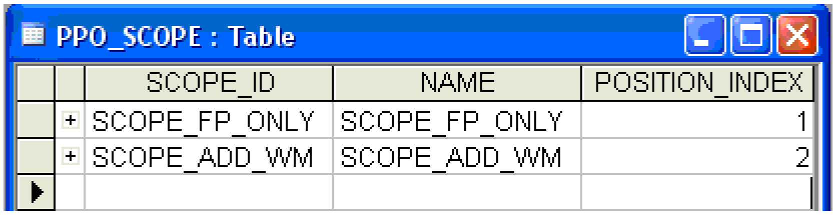
PPO_RECIPE_FAMILY : Table

RECIPE_FAMILY_ID	NAME	TYPE
FP	FP	PROCESS_STAGE
WM	WM	PROCESS_STAGE

The next table is the PPO_RECIPE_RECIPE_FAMILY table and it is used to organize the recipes into their appropriate recipe families. Note that a recipe can belong to only one recipe family.

Next you must create the *scopes*. Scopes are used to control which part of the manufacturing process that you intend to solve; in other words, which part of the decomposition is solved. A scope can contain several recipe families, each with a different status (frozen or planned).

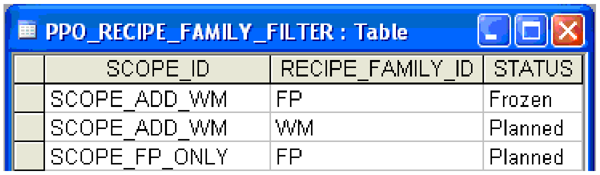
In this example, we create one scope for the finished products and one for the white mass. As we are using push demand and want the finished products scheduled first, the SCOPE_FP_ONLY gets a POSITION_INDEX of "1" and the white mass is second in position.



PPO_SCOPE : Table

SCOPE_ID	NAME	POSITION_INDEX
SCOPE_FP_ONLY	SCOPE_FP_ONLY	1
SCOPE_ADD_WM	SCOPE_ADD_WM	2

The next table is used to associate the recipe families with a scope, and to indicate a status of frozen or planned. Recipe families can belong to more than one scope, as necessary for the decomposition. The status of **Planned** means that the orders of that recipe family are to be planned when this scope is solved. The status of **Frozen** means that the orders of the recipe family are fixed — they cannot be changed or moved during the solve. Therefore these recipe orders act as a component entry constraint or as a dependent material requirement.

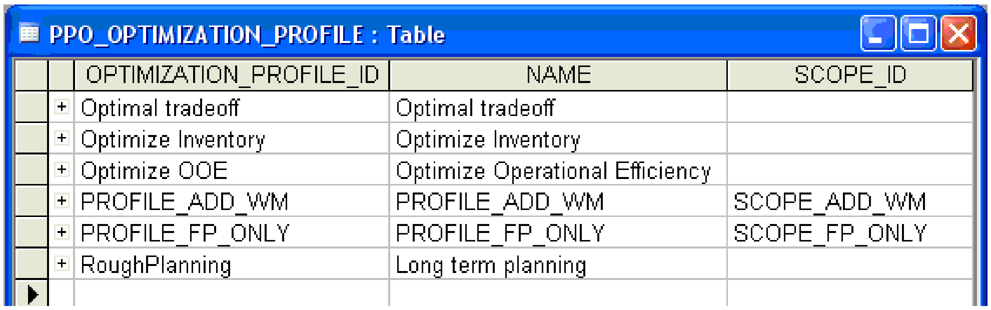


PPO_RECIPE_FAMILY_FILTER : Table

SCOPE_ID	RECIPE_FAMILY_ID	STATUS
SCOPE_ADD_WM	FP	Frozen
SCOPE_ADD_WM	WM	Planned
SCOPE_FP_ONLY	FP	Planned

From the PPO_SCOPE table, you know that the first scope to be solved in a decomposition is the SCOPE_FP_ONLY, containing only finished products recipes. This scope creates a schedule of finished products depending on the real demand for those products, regardless of any constraints on white mass availability. Next the SCOPE_ADD_WM is solved, which plans the manufacture of semi-finished white mass to balance the already created and frozen finished product orders.

Once you create the scopes, you must associate them with an optimization profile to make the scopes available to the solve process. An optimization profile contains particular methods of solving, such as algorithms and parameters. You may even want to have two optimization profiles solving with the same scope but with differing algorithms or parameters, but in this example each scope is associated with only one profile. The following table makes the scope available for use in the PPO GUI.



PPO_OPTIMIZATION_PROFILE : Table

OPTIMIZATION_PROFILE_ID	NAME	SCOPE_ID
Optimal tradeoff	Optimal tradeoff	
Optimize Inventory	Optimize Inventory	
Optimize OOE	Optimize Operational Efficiency	
PROFILE_ADD_WM	PROFILE_ADD_WM	SCOPE_ADD_WM
PROFILE_FP_ONLY	PROFILE_FP_ONLY	SCOPE_FP_ONLY
RoughPlanning	Long term planning	

Optimization using scopes in the GUI

The scope appears in the **Optimize the scenario** dialog box which displays when you optimize the problem. The scope appears nowhere else in the GUI. It is associated with the optimization profile set in the PPO_OPTIMIZATION_PROFILE table.

In the previous example, to solve the entire decomposition problem, you would first select PROFILE_FP_ONLY and optimize the scenario; this results in a schedule of finished products. Next select PROFILE_ADD_WM and optimize. This adds a schedule for white mass manufacture, which corresponds to the requirements to make the finished products. The result is a full schedule of white mass and finished product.

Advanced usage of the decomposition framework using Java API

If you plug your own algorithm in PPO you may use the Java™ API to help decompose the problem. This API simplifies the submodel creation and the reporting of results of a submodel on a master model.

Example of decomposition with a two step optimization; first finished goods then semi-finished products are scheduled.

```
class MyEngineOptimizer extends IloMSEngineOptimizer

{

boolean solveSubModel()

{

  IloMSModel subModel = scope.buildSubModel();

  boolean solved      = subModel.solve();

  if(solved)

     scope.transferResults(subModel, scope.getModel());

  subModel.end();

  return solved;

}


boolean solve(IloMSModel masterModel)

{

  IloMSScope scopeFP = model.getScopeByIdentifier("SCOPE_FP_ONLY");

  boolean result = solveSubModel(scopeFP);

  if (result) {

    IloMSScope scopeWM = model.getScopeByIdentifier("SCOPE_ADD_WM");

    boolean solved = solveSubModel(scopeWM);

    result = result && solved;

  }

  return result;
}

}
```

Such a class must be instantiated by a subclass of **IloMSOptimizerFactory**.

```
class MyOptimizerFactory extends IloMSOptimizerFactory

{

  public IloMSEngineOptimizer create()

  {

    return new MyEngineOptimizer();

  }

 }
```

The singleton of the subclass must be set in the plug-in initialization code.

```
IloMSOptimizerFactory.setFactory(new MyOptimizerfactory());
```

Production planning simulations

The planning solution is editable in the PPO GUI and allows you to run simulations and re-optimize with planning.

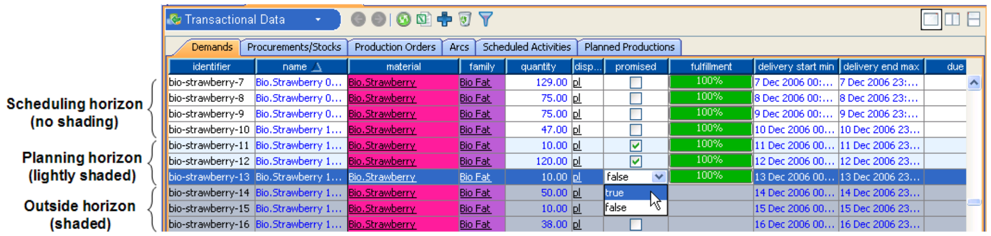
In this section

Interactive planning
Describes some of the views that enable you to run planning simulations.

Interactive planning

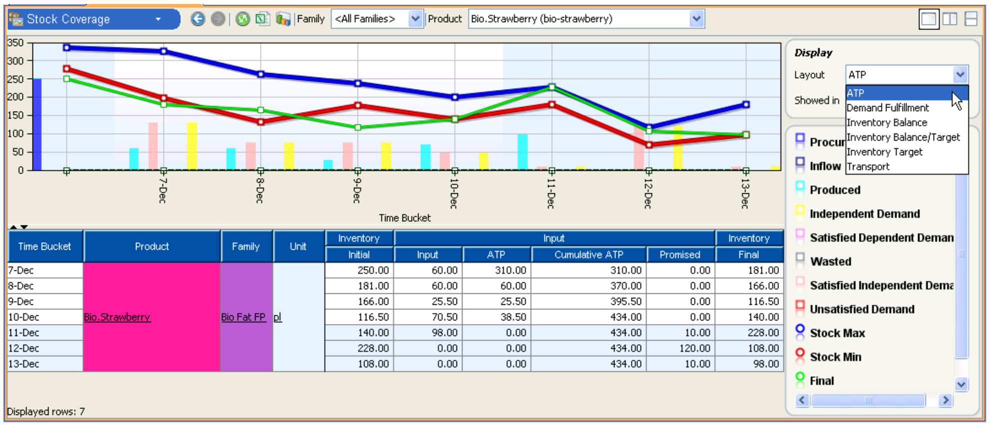
The planning solution is editable. This means that you can freeze sections of the plan, make simulations, and re-optimize with the planning engine. You can set *promised* demand representing material that is already promised to a customer, then re-optimize to determine if you can deliver more product, how much it would cost, and how much revenue you would receive.

The following image shows the **Demands** tab of the **Transactional Data** view. Note that the **fulfillment** column shows **100%** for all demands within the scheduling and planning horizons. The **promised** field can be set to **true** in preparation to run a simulation to determine how much more demand could be met while keeping the current demands fulfilled.



identifier	name	material	family	quantity	disp...	promised	fulfillment	delivery start min	delivery end max	due
bio-strawberry-7	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	129.00	pl	☐	100%	7 Dec 2006 00:...	7 Dec 2006 23:...	
bio-strawberry-8	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	75.00	pl	☐	100%	8 Dec 2006 00:...	8 Dec 2006 23:...	
bio-strawberry-9	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	75.00	pl	☐	100%	9 Dec 2006 00:...	9 Dec 2006 23:...	
bio-strawberry-10	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	47.00	pl	☐	100%	10 Dec 2006 00...	10 Dec 2006 23...	
bio-strawberry-11	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	10.00	pl	✓	100%	11 Dec 2006 00...	11 Dec 2006 23...	
bio-strawberry-12	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	120.00	pl	✓	100%	12 Dec 2006 00...	12 Dec 2006 23...	
bio-strawberry-13	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	10.00	pl	false	100%	13 Dec 2006 00...	13 Dec 2006 23...	
bio-strawberry-14	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	50.00	pl	true		14 Dec 2006 00...	14 Dec 2006 23...	
bio-strawberry-15	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	10.00	pl	false		15 Dec 2006 00...	15 Dec 2006 23...	
bio-strawberry-16	Bio.Strawberry 1...	Bio.Strawberry	Bio Fat	38.00	pl	☐		16 Dec 2006 00...	16 Dec 2006 23...	

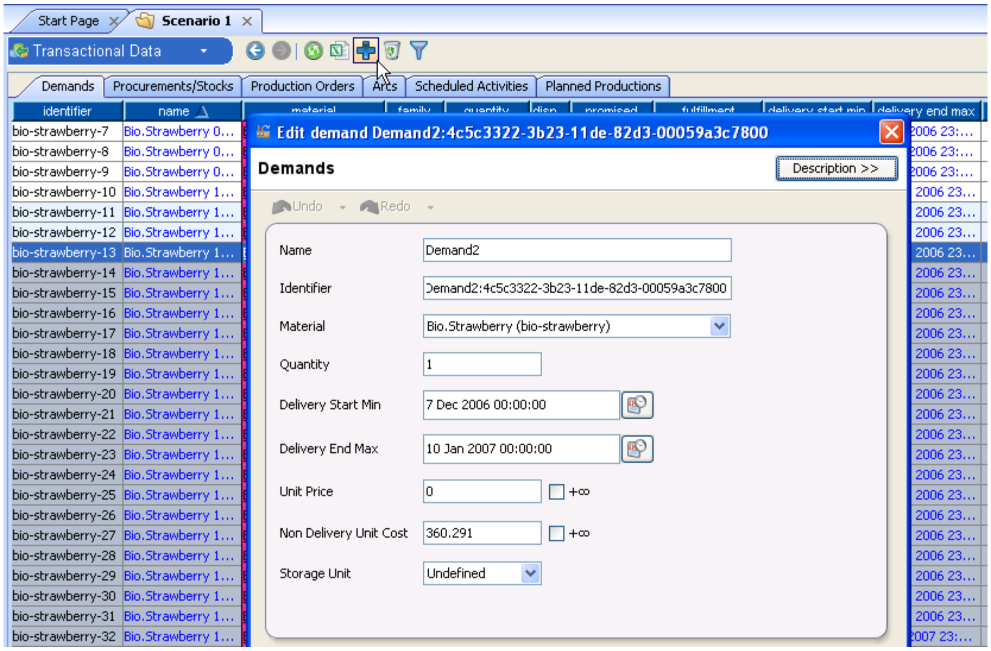
On the **Stock Coverage** view, the **ATP** layout reveals the current values for the **ATP**, **Cumulative ATP**, and **Promised** amounts. The available-to-promise field shows the extra amount that could be produced for that day; the cumulative field shows the total amount as time progresses. The promised value shown below reflects the **promised** column of the **Transactional Data** view, above. Without changing the planning solution, we can promise an additional 310 units on December 7.



Time Bucket	Product	Family	Unit	Inventory	Input				Inventory
				Initial	Input	ATP	Cumulative ATP	Promised	Final
7-Dec	Bio.Strawberry	Bio Fat FP	pl	250.00	60.00	310.00	310.00	0.00	181.00
8-Dec				181.00	60.00	60.00	370.00	0.00	166.00
9-Dec				166.00	25.50	25.50	395.50	0.00	116.50
10-Dec				116.50	70.50	38.50	434.00	0.00	140.00
11-Dec				140.00	98.00	0.00	434.00	10.00	228.00
12-Dec				228.00	0.00	0.00	434.00	120.00	108.00
13-Dec				108.00	0.00	0.00	434.00	10.00	98.00

Displayed rows: 7

You can add more demands on the **Demands** tab, using the plus icon in the toolbar. The **Demands** editing window displays, allowing you to specify the name, quantity, delivery time window, non delivery cost and revenue per unit of satisfied demand (unit price). This data will help you determine if it is possible to deliver more material, at what cost, and with what income.



identifier	name	delivery end max
bio-strawberry-7	Bio.Strawberry 0…	2006 23:…
bio-strawberry-8	Bio.Strawberry 0…	2006 23:…
bio-strawberry-9	Bio.Strawberry 0…	2006 23:…
bio-strawberry-10	Bio.Strawberry 1…	2006 23…
bio-strawberry-11	Bio.Strawberry 1…	2006 23…
bio-strawberry-12	Bio.Strawberry 1…	2006 23…
bio-strawberry-13	Bio.Strawberry 1…	2006 23…
bio-strawberry-14	Bio.Strawberry 1…	2006 23…
bio-strawberry-15	Bio.Strawberry 1…	2006 23…
bio-strawberry-16	Bio.Strawberry 1…	2006 23…
bio-strawberry-17	Bio.Strawberry 1…	2006 23…
bio-strawberry-18	Bio.Strawberry 1…	2006 23…
bio-strawberry-19	Bio.Strawberry 1…	2006 23…
bio-strawberry-20	Bio.Strawberry 1…	2006 23…
bio-strawberry-21	Bio.Strawberry 1…	2006 23…
bio-strawberry-22	Bio.Strawberry 1…	2006 23…
bio-strawberry-23	Bio.Strawberry 1…	2006 23…
bio-strawberry-24	Bio.Strawberry 1…	2006 23…
bio-strawberry-25	Bio.Strawberry 1…	2006 23…
bio-strawberry-26	Bio.Strawberry 1…	2006 23…
bio-strawberry-27	Bio.Strawberry 1…	2006 23…
bio-strawberry-28	Bio.Strawberry 1…	2006 23…
bio-strawberry-29	Bio.Strawberry 1…	2006 23…
bio-strawberry-30	Bio.Strawberry 1…	2006 23…
bio-strawberry-31	Bio.Strawberry 1…	2006 23…
bio-strawberry-32	Bio.Strawberry 1…	2007 23:…

Name	Demand2
Identifier	Demand2:4c5c3322-3b23-11de-82d3-00059a3c7800
Material	Bio.Strawberry (bio-strawberry)
Quantity	1
Delivery Start Min	7 Dec 2006 00:00:00
Delivery End Max	10 Jan 2007 00:00:00
Unit Price	0 ☐ +∞
Non Delivery Unit Cost	360.291 ☐ +∞
Storage Unit	Undefined

Re-optimize the scenario, using only the planning module; uncheck the batching and scheduling modules. Be careful to set the cost and revenue weights on the **Planning advanced options** window to appropriate values for this problem.

Once optimization is complete, check the **Transactional Data** and **Stock Coverage** views to see how much of the new demand was fulfilled. The **KPIs Summary** view shows the new costs and revenue.

On the **Planned Productions** tab in the **Transactional Data** view, you can set a firm minimum for the production plans of the current solution.

Advanced usage: Distribution planning

This advanced section describes how to use PPO to optimize your distribution network. Any usage of PPO for distribution planning, supply chain planning, or multi-plant planning must be validated with the PPO product management team.

In this section

General concepts
Introduces the terms master planning and distribution planning.

The distribution plan calculation
Describes factors that affect the creation of the distribution plan.

Modeling the distribution plan
An example of using the PPO data schema to model the plan.

General concepts

Distribution planning is a process that includes deciding how and where to produce materials, what materials to transport and how to move them, and what materials to stock in which locations.

Master planning is the general process that features:

- ♦ A long time horizon (year) with large time buckets (weekly or monthly);
- ♦ Materials aggregated by family type;
- ♦ Use of macro-resources;
- ♦ Decisions of how much and where to produce materials;
- ♦ Decisions of how much and where to stock materials;
- ♦ Decisions of how to transport materials;

Distribution planning is the optimization of the master planning, based on costs:

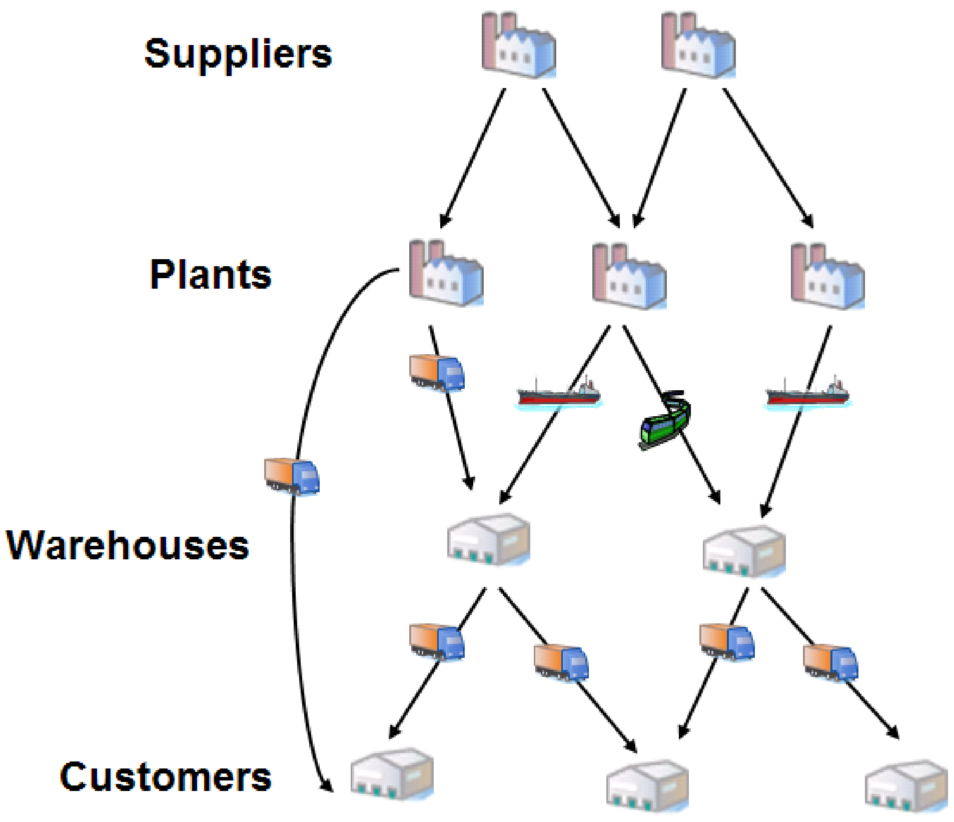
- ♦ Production volumes;
- ♦ Allocated capacities for production, inventories, and transportation;
- ♦ Stock objectives;
- ♦ Transportation modes.

The distribution network is composed of four types of sites:

- ♦ **Suppliers** that ship raw materials and raw packaging;
- ♦ **Plants** that are production sites transforming raw materials into finished products;
- ♦ **Warehouses** that are intermediate sites for receiving, stocking and expediting.
- ♦ **Customers** or **sales points** to receive products or sell to final customers.

Customers can actually represent a global market (demand of a region or country) and may also be attached to plants or warehouses.

The distribution network is represented in the following graphic.


Suppliers
Plants
Warehouses
Customers

The distribution plan calculation

The purpose of the distribution plan is to control the flow of materials through your network; to track this you need to implement the Stock Keeping Unit (SKU) which is a matched pair of product and site (location).

The input data for the distribution plan includes:

- stock levels
- macro-demands by marketplace
- costs of production
- costs of transportation.

The goal of the plan is to provide an estimate on:

- global supplies
- production from each plant
- resource levels
- arcs chosen in the network
- transportation mode
- stock objectives.

Complexity considerations include:

- size of data
- SKU numbers
- number of sites
- number of flow arcs
- vehicle fulfillment on transportation modes (not recommended as it's probably not relevant at strategic level).

Modeling the distribution plan

An example of using the PPO data schema to model the plan.

In this section

Key modeling tables
Describes the primary schema tables for modeling a distribution network.

Building a distribution plan model
Using the example file beer_distribution_planning.csv.

Viewing the results
Examines the planning results from the example.

Key modeling tables

The main schema tables for distribution planning are:

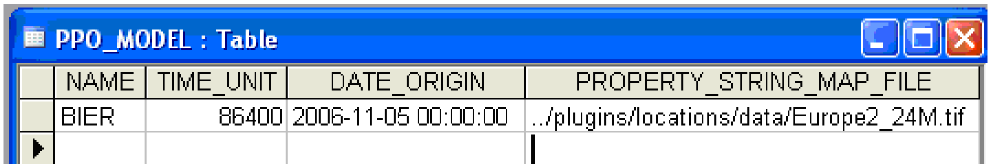
- PPO_MATERIAL used to model the SKUs.
- PPO_STORAGE_UNIT used to model the sites — the plants and warehouses of the network.
- PPO_RESOURCE to model production and transportation resources.
- PPO_RECIPE to model production and transportation recipes.
- PPO_MATERIAL_PRODUCTION is associated with transportation recipes in the consumption of material from a storage unit of the incoming site and production to the storage unit of the receptive site.
- PPO_DEMAND used to detail demands by storage unit (equivalent to demand by site).

Building a distribution plan model

This section describes the key modeling tables used in the example file `beer_distribution_planning.csv` to model a distribution network.

Time unit

In a planning network we are typically dealing with a long horizon and large time buckets, so we can define a 24-hour day as the time unit (86400 seconds) in the PPO_MODEL table.



PPO_MODEL : Table

NAME	TIME_UNIT	DATE_ORIGIN	PROPERTY_STRING_MAP_FILE
BIER	86400	2006-11-05 00:00:00	../plugins/locations/data/Europe2_24M.tif

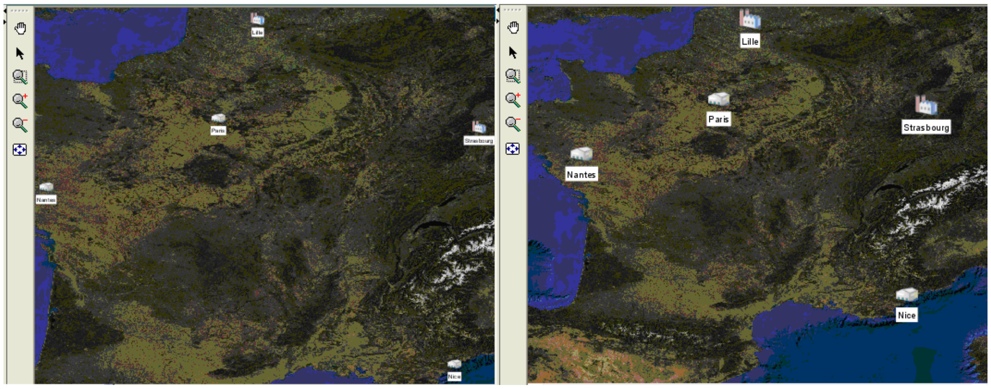
Map graphic for the Planning View

In the PPO_SETTING table you define a map file used for display in the **Distribution Planning** view of the PPO GUI:

```
PROPERTY_STRING_MAP_FILE ../plugins/locations/data/Europe2_24M.tif
```

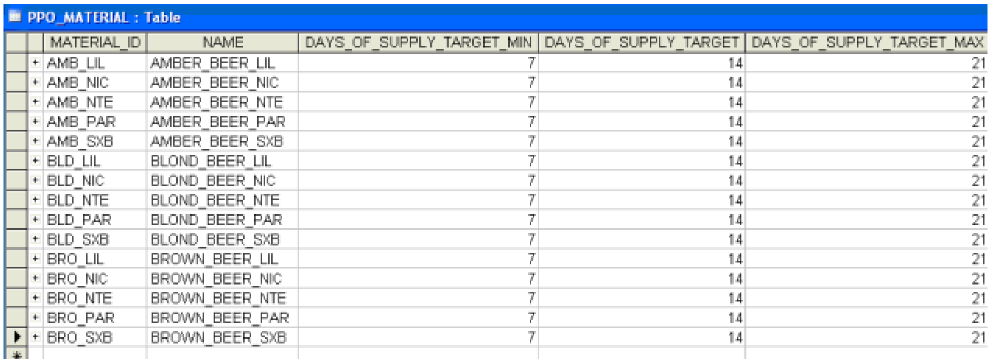
This map graphic should be a georeferenced TIFF file (GeoTIFF) so that it can handle latitude and longitude information.

Another setting called the *Map Graphic Factor* is needed to specify the zoom factor on the sites and links shown on the map. This needs to be adjusted depending on the area covered by the sites and the distances between sites. The following figure shows the same graphic with a `MapGraphicFactor` of 0.01 on the left, and on the right a `MapGraphicFactor` of 0.02.



Materials

Materials are described with their targets. In this example, the amber beer at Lille is an SKU and the amber beer in Paris is another SKU.



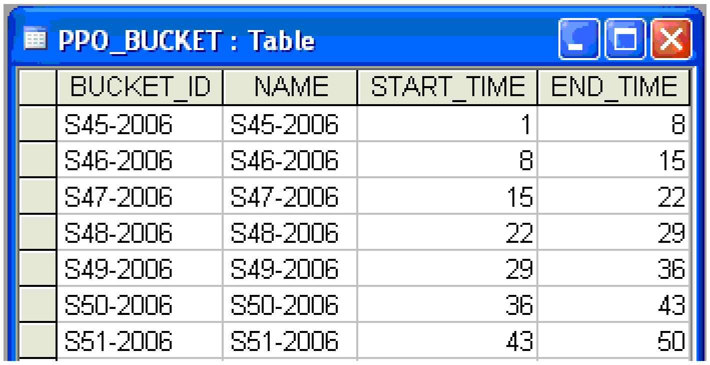
PPO_MATERIAL : Table

MATERIAL_ID	NAME	DAYS_OF_SUPPLY_TARGET_MIN	DAYS_OF_SUPPLY_TARGET	DAYS_OF_SUPPLY_TARGET_MAX
AMB_LIL	AMBER_BEER_LIL	7	14	21
AMB_NIC	AMBER_BEER_NIC	7	14	21
AMB_NTE	AMBER_BEER_NTE	7	14	21
AMB_PAR	AMBER_BEER_PAR	7	14	21
AMB_SXB	AMBER_BEER_SXB	7	14	21
BLD_LIL	BLOND_BEER_LIL	7	14	21
BLD_NIC	BLOND_BEER_NIC	7	14	21
BLD_NTE	BLOND_BEER_NTE	7	14	21
BLD_PAR	BLOND_BEER_PAR	7	14	21
BLD_SXB	BLOND_BEER_SXB	7	14	21
BRO_LIL	BROWN_BEER_LIL	7	14	21
BRO_NIC	BROWN_BEER_NIC	7	14	21
BRO_NTE	BROWN_BEER_NTE	7	14	21
BRO_PAR	BROWN_BEER_PAR	7	14	21
BRO_SXB	BROWN_BEER_SXB	7	14	21

Typically it's preferable to plan families of products to reduce complexity; this is a strategic decision.

Time buckets

The horizon is quite long so the planning buckets should be large also. This example uses a seven-day long bucket.

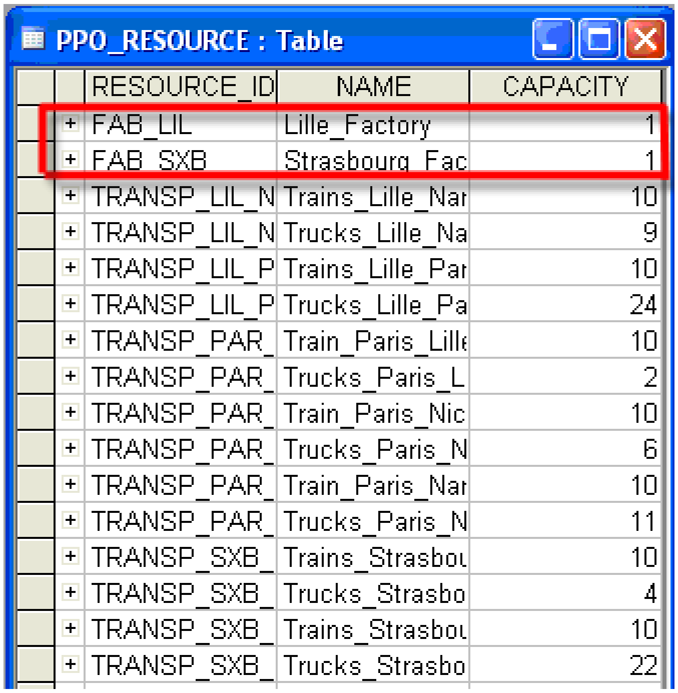


PPO_BUCKET : Table

BUCKET_ID	NAME	START_TIME	END_TIME
S45-2006	S45-2006	1	8
S46-2006	S46-2006	8	15
S47-2006	S47-2006	15	22
S48-2006	S48-2006	22	29
S49-2006	S49-2006	29	36
S50-2006	S50-2006	36	43
S51-2006	S51-2006	43	50

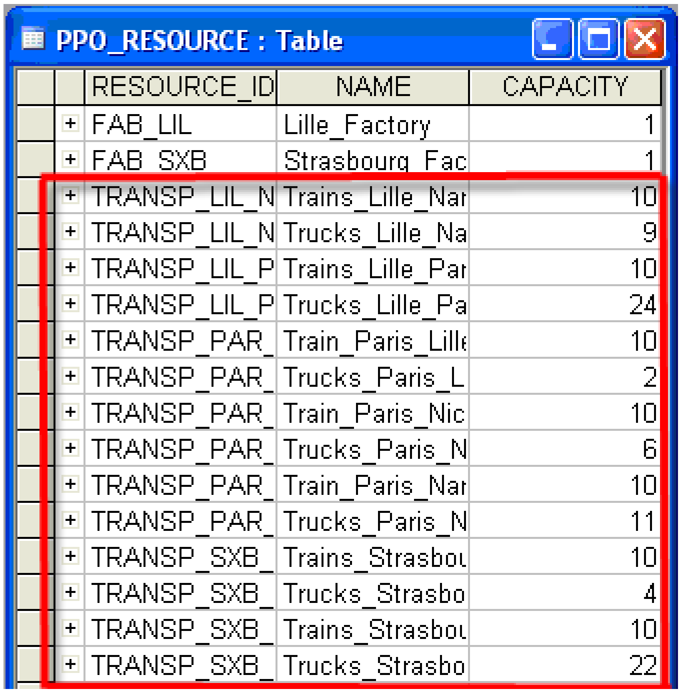
Resources

This example uses production and transportation resources. The production resources are macro-resources in this case; the two plants involved. The transportation resources model the various modes of transportation between sites along with the transport capacities. The first table highlights the production resources, and the second highlights the transport resources.



PPO_RESOURCE : Table

		RESOURCE_ID	NAME	CAPACITY
	+	FAB_LIL	Lille_Factory	1
	+	FAB_SXB	Strasbourg_Fac	1
	+	TRANSP_LIL_N	Trains_Lille_Nar	10
	+	TRANSP_LIL_N	Trucks_Lille_Na	9
	+	TRANSP_LIL_P	Trains_Lille_Par	10
	+	TRANSP_LIL_P	Trucks_Lille_Pa	24
	+	TRANSP_PAR_	Train_Paris_Lille	10
	+	TRANSP_PAR_	Trucks_Paris_L	2
	+	TRANSP_PAR_	Train_Paris_Nic	10
	+	TRANSP_PAR_	Trucks_Paris_N	6
	+	TRANSP_PAR_	Train_Paris_Nar	10
	+	TRANSP_PAR_	Trucks_Paris_N	11
	+	TRANSP_SXB_	Trains_Strasbou	10
	+	TRANSP_SXB_	Trucks_Strasbo	4
	+	TRANSP_SXB_	Trains_Strasbou	10
	+	TRANSP_SXB_	Trucks_Strasbo	22

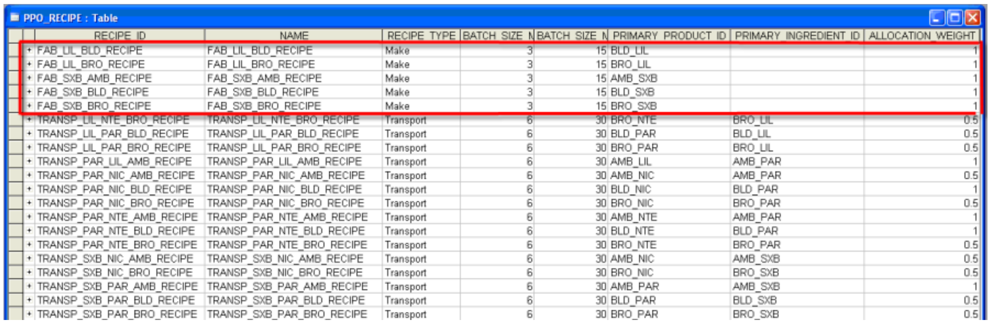

PPO_RESOURCE : Table

RESOURCE_ID	NAME	CAPACITY
FAB_LIL	Lille_Factory	1
FAB_SXB	Strasbourg_Fac	1
TRANSP_LIL_N	Trains_Lille_Nar	10
TRANSP_LIL_N	Trucks_Lille_Na	9
TRANSP_LIL_P	Trains_Lille_Par	10
TRANSP_LIL_P	Trucks_Lille_Pa	24
TRANSP_PAR_	Train_Paris_Lille	10
TRANSP_PAR_	Trucks_Paris_L	2
TRANSP_PAR_	Train_Paris_Nic	10
TRANSP_PAR_	Trucks_Paris_N	6
TRANSP_PAR_	Train_Paris_Nar	10
TRANSP_PAR_	Trucks_Paris_N	11
TRANSP_SXB_	Trains_Strasbou	10
TRANSP_SXB_	Trucks_Strasbo	4
TRANSP_SXB_	Trains_Strasbou	10
TRANSP_SXB_	Trucks_Strasbo	22

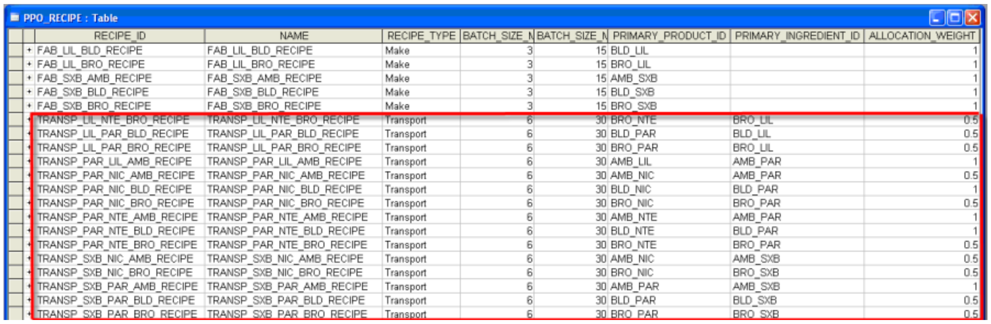
Recipes

There are production and transportation recipes. The PRIMARY_PRODUCT_ID of the recipe identifies the material that is produced or transported. The RECIPE_TYPE identifies the recipe as production or transportation; this distinction is important to separate the production produced from the production transported as reported in the GUI views. However the optimizer does not make any distinction between these two type of recipes.

For convenience you can incorporate the name of the production site in the production RECIPE_ID (first diagram) and the name of the two involved sites in the transportation RECIPE_ID (second diagram).


PPO_RECIPE : Table

RECIPE_ID	NAME	RECIPE_TYPE	BATCH_SIZE_N	BATCH_SIZE_N	PRIMARY_PRODUCT_ID	PRIMARY_INGREDIENT_ID	ALLOCATION_WEIGHT
FAB_LIL_BLD_RECIPE	FAB_LIL_BLD_RECIPE	Make	3	15	BLD_LIL		1
FAB_LIL_BRO_RECIPE	FAB_LIL_BRO_RECIPE	Make	3	15	BRO_LIL		1
FAB_SXB_AMB_RECIPE	FAB_SXB_AMB_RECIPE	Make	3	15	AMB_SXB		1
FAB_SXB_BLD_RECIPE	FAB_SXB_BLD_RECIPE	Make	3	15	BLD_SXB		1
FAB_SXB_BRO_RECIPE	FAB_SXB_BRO_RECIPE	Make	3	15	BRO_SXB		1
TRANSP_LIL_NTE_BRO_RECIPE	TRANSP_LIL_NTE_BRO_RECIPE	Transport	6	30	BRO_NTE	BRO_LIL	0.5
TRANSP_LIL_PAR_BLD_RECIPE	TRANSP_LIL_PAR_BLD_RECIPE	Transport	6	30	BLD_PAR	BLD_LIL	0.5
TRANSP_LIL_PAR_BRO_RECIPE	TRANSP_LIL_PAR_BRO_RECIPE	Transport	6	30	BRO_PAR	BRO_LIL	0.5
TRANSP_PAR_LIL_AMB_RECIPE	TRANSP_PAR_LIL_AMB_RECIPE	Transport	6	30	AMB_LIL	AMB_PAR	1
TRANSP_PAR_NIC_AMB_RECIPE	TRANSP_PAR_NIC_AMB_RECIPE	Transport	6	30	AMB_NIC	AMB_PAR	0.5
TRANSP_PAR_NIC_BLD_RECIPE	TRANSP_PAR_NIC_BLD_RECIPE	Transport	6	30	BLD_NIC	BLD_PAR	1
TRANSP_PAR_NIC_BRO_RECIPE	TRANSP_PAR_NIC_BRO_RECIPE	Transport	6	30	BRO_NIC	BRO_PAR	0.5
TRANSP_PAR_NTE_AMB_RECIPE	TRANSP_PAR_NTE_AMB_RECIPE	Transport	6	30	AMB_NTE	AMB_PAR	1
TRANSP_PAR_NTE_BLD_RECIPE	TRANSP_PAR_NTE_BLD_RECIPE	Transport	6	30	BLD_NTE	BLD_PAR	1
TRANSP_PAR_NTE_BRO_RECIPE	TRANSP_PAR_NTE_BRO_RECIPE	Transport	6	30	BRO_NTE	BRO_PAR	0.5
TRANSP_SXB_NIC_AMB_RECIPE	TRANSP_SXB_NIC_AMB_RECIPE	Transport	6	30	AMB_NIC	AMB_SXB	0.5
TRANSP_SXB_NIC_BRO_RECIPE	TRANSP_SXB_NIC_BRO_RECIPE	Transport	6	30	BRO_NIC	BRO_SXB	0.5
TRANSP_SXB_PAR_AMB_RECIPE	TRANSP_SXB_PAR_AMB_RECIPE	Transport	6	30	AMB_PAR	AMB_SXB	1
TRANSP_SXB_PAR_BLD_RECIPE	TRANSP_SXB_PAR_BLD_RECIPE	Transport	6	30	BLD_PAR	BLD_SXB	0.5
TRANSP_SXB_PAR_BRO_RECIPE	TRANSP_SXB_PAR_BRO_RECIPE	Transport	6	30	BRO_PAR	BRO_SXB	0.5

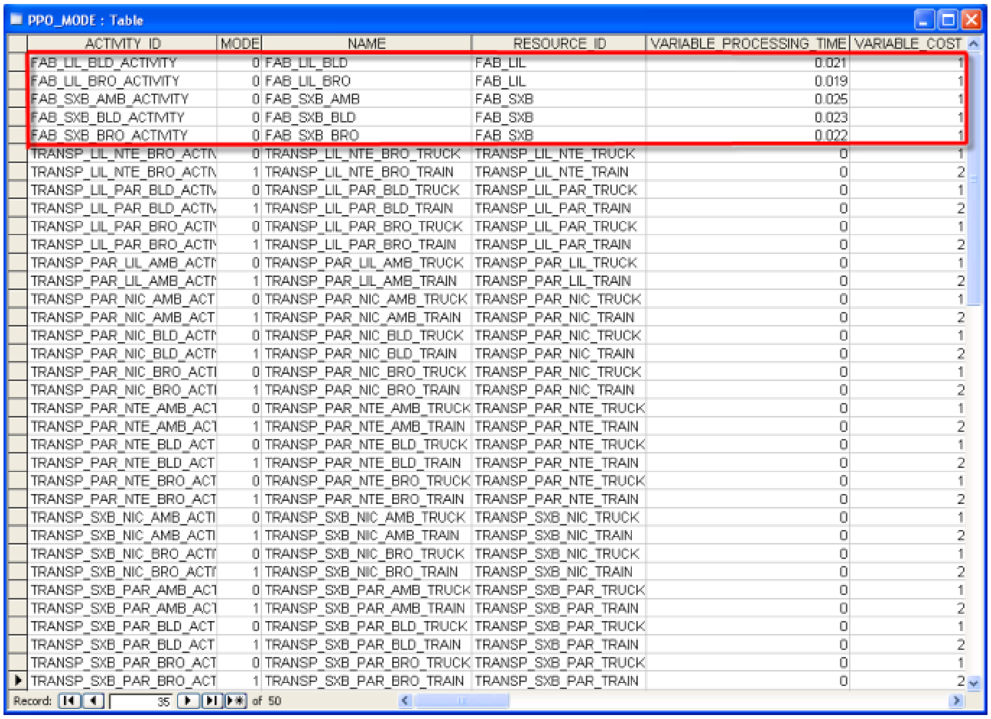


PPO_RECIPE : Table

RECIPE_ID	NAME	RECIPE_TYPE	BATCH_SIZE_N	BATCH_SIZE_N	PRIMARY_PRODUCT_ID	PRIMARY_INGREDIENT_ID	ALLOCATION_WEIGHT
FAB_LIL_BLD_RECIPE	FAB_LIL_BLD_RECIPE	Make	3	15	BLD_LIL		1
FAB_LIL_BRO_RECIPE	FAB_LIL_BRO_RECIPE	Make	3	15	BRO_LIL		1
FAB_SXB_AMB_RECIPE	FAB_SXB_AMB_RECIPE	Make	3	15	AMB_SXB		1
FAB_SXB_BLD_RECIPE	FAB_SXB_BLD_RECIPE	Make	3	15	BLD_SXB		1
FAB_SXB_BRO_RECIPE	FAB_SXB_BRO_RECIPE	Make	3	15	BRO_SXB		1
TRANSP_LIL_NTE_BRO_RECIPE	TRANSP_LIL_NTE_BRO_RECIPE	Transport	6	30	BRO_NTE	BRO_LIL	0.5
TRANSP_LIL_PAR_BLD_RECIPE	TRANSP_LIL_PAR_BLD_RECIPE	Transport	6	30	BLD_PAR	BLD_LIL	0.5
TRANSP_LIL_PAR_BRO_RECIPE	TRANSP_LIL_PAR_BRO_RECIPE	Transport	6	30	BRO_PAR	BRO_LIL	0.5
TRANSP_PAR_LIL_AMB_RECIPE	TRANSP_PAR_LIL_AMB_RECIPE	Transport	6	30	AMB_LIL	AMB_PAR	1
TRANSP_PAR_NIC_AMB_RECIPE	TRANSP_PAR_NIC_AMB_RECIPE	Transport	6	30	AMB_NIC	AMB_PAR	0.5
TRANSP_PAR_NIC_BLD_RECIPE	TRANSP_PAR_NIC_BLD_RECIPE	Transport	6	30	BLD_NIC	BLD_PAR	1
TRANSP_PAR_NIC_BRO_RECIPE	TRANSP_PAR_NIC_BRO_RECIPE	Transport	6	30	BRO_NIC	BRO_PAR	0.5
TRANSP_PAR_NTE_AMB_RECIPE	TRANSP_PAR_NTE_AMB_RECIPE	Transport	6	30	AMB_NTE	AMB_PAR	1
TRANSP_PAR_NTE_BLD_RECIPE	TRANSP_PAR_NTE_BLD_RECIPE	Transport	6	30	BLD_NTE	BLD_PAR	1
TRANSP_PAR_NTE_BRO_RECIPE	TRANSP_PAR_NTE_BRO_RECIPE	Transport	6	30	BRO_NTE	BRO_PAR	0.5
TRANSP_SXB_NIC_AMB_RECIPE	TRANSP_SXB_NIC_AMB_RECIPE	Transport	6	30	AMB_NIC	AMB_SXB	0.5
TRANSP_SXB_NIC_BRO_RECIPE	TRANSP_SXB_NIC_BRO_RECIPE	Transport	6	30	BRO_NIC	BRO_SXB	0.5
TRANSP_SXB_PAR_AMB_RECIPE	TRANSP_SXB_PAR_AMB_RECIPE	Transport	6	30	AMB_PAR	AMB_SXB	1
TRANSP_SXB_PAR_BLD_RECIPE	TRANSP_SXB_PAR_BLD_RECIPE	Transport	6	30	BLD_PAR	BLD_SXB	0.5
TRANSP_SXB_PAR_BRO_RECIPE	TRANSP_SXB_PAR_BRO_RECIPE	Transport	6	30	BRO_PAR	BRO_SXB	0.5

Modes

Modes represent the different methods or ways of producing and transporting the materials; for the same activity, different times, costs and resources may be involved. For example, the differing VARIABLE_PROCESSING_TIME values for the beer production activities and the RESOURCE_IDs used are shown in the following image.

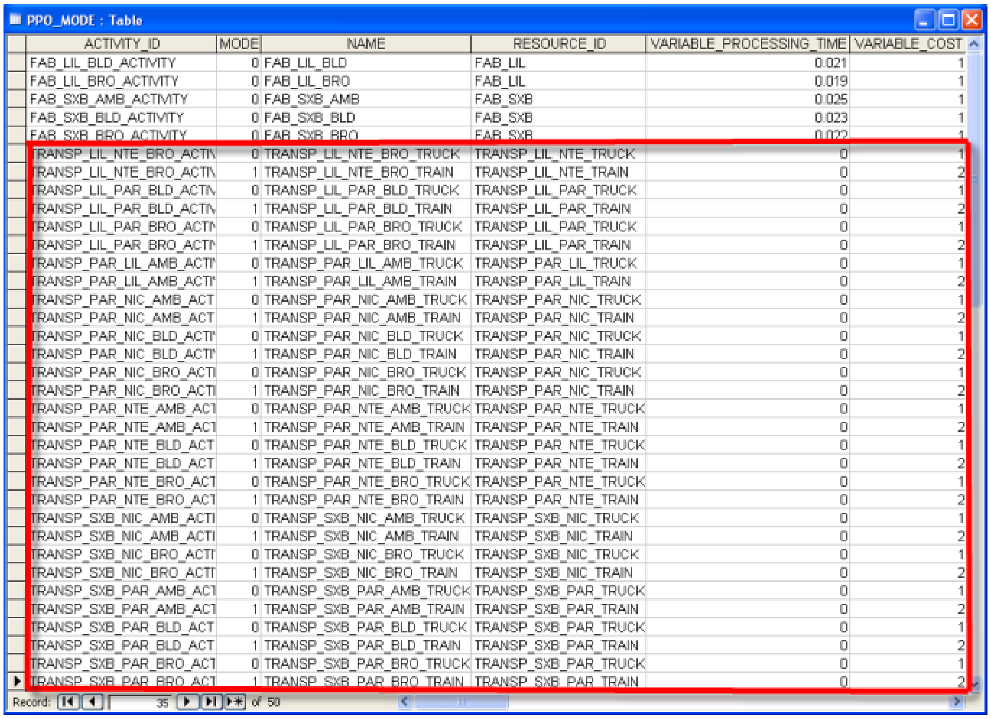


PPO_MODE : Table

ACTIVITY_ID	MODE	NAME	RESOURCE_ID	VARIABLE_PROCESSING_TIME	VARIABLE_COST
FAB_LIL_BLD_ACTIVITY	0	FAB_LIL_BLD	FAB_LIL	0.021	1
FAB_LIL_BRO_ACTIVITY	0	FAB_LIL_BRO	FAB_LIL	0.019	1
FAB_SXB_AMB_ACTIVITY	0	FAB_SXB_AMB	FAB_SXB	0.025	1
FAB_SXB_BLD_ACTIVITY	0	FAB_SXB_BLD	FAB_SXB	0.023	1
FAB_SXB_BRO_ACTIVITY	0	FAB_SXB_BRO	FAB_SXB	0.022	1
TRANSP_LIL_NTE_BRO_ACTI	0	TRANSP_LIL_NTE_BRO_TRUCK	TRANSP_LIL_NTE_TRUCK	0	1
TRANSP_LIL_NTE_BRO_ACTI	1	TRANSP_LIL_NTE_BRO_TRAIN	TRANSP_LIL_NTE_TRAIN	0	2
TRANSP_LIL_PAR_BLD_ACTI	0	TRANSP_LIL_PAR_BLD_TRUCK	TRANSP_LIL_PAR_TRUCK	0	1
TRANSP_LIL_PAR_BLD_ACTI	1	TRANSP_LIL_PAR_BLD_TRAIN	TRANSP_LIL_PAR_TRAIN	0	2
TRANSP_LIL_PAR_BRO_ACTI	0	TRANSP_LIL_PAR_BRO_TRUCK	TRANSP_LIL_PAR_TRUCK	0	1
TRANSP_LIL_PAR_BRO_ACTI	1	TRANSP_LIL_PAR_BRO_TRAIN	TRANSP_LIL_PAR_TRAIN	0	2
TRANSP_PAR_LIL_AMB_ACTI	0	TRANSP_PAR_LIL_AMB_TRUCK	TRANSP_PAR_LIL_TRUCK	0	1
TRANSP_PAR_LIL_AMB_ACTI	1	TRANSP_PAR_LIL_AMB_TRAIN	TRANSP_PAR_LIL_TRAIN	0	2
TRANSP_PAR_NIC_AMB_ACT	0	TRANSP_PAR_NIC_AMB_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_AMB_ACT	1	TRANSP_PAR_NIC_AMB_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NIC_BLD_ACTI	0	TRANSP_PAR_NIC_BLD_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_BLD_ACTI	1	TRANSP_PAR_NIC_BLD_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NIC_BRO_ACTI	0	TRANSP_PAR_NIC_BRO_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_BRO_ACTI	1	TRANSP_PAR_NIC_BRO_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NTE_AMB_ACT	0	TRANSP_PAR_NTE_AMB_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_AMB_ACT	1	TRANSP_PAR_NTE_AMB_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_PAR_NTE_BLD_ACT	0	TRANSP_PAR_NTE_BLD_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_BLD_ACT	1	TRANSP_PAR_NTE_BLD_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_PAR_NTE_BRO_ACT	0	TRANSP_PAR_NTE_BRO_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_BRO_ACT	1	TRANSP_PAR_NTE_BRO_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_SXB_NIC_AMB_ACTI	0	TRANSP_SXB_NIC_AMB_TRUCK	TRANSP_SXB_NIC_TRUCK	0	1
TRANSP_SXB_NIC_AMB_ACTI	1	TRANSP_SXB_NIC_AMB_TRAIN	TRANSP_SXB_NIC_TRAIN	0	2
TRANSP_SXB_NIC_BRO_ACTI	0	TRANSP_SXB_NIC_BRO_TRUCK	TRANSP_SXB_NIC_TRUCK	0	1
TRANSP_SXB_NIC_BRO_ACTI	1	TRANSP_SXB_NIC_BRO_TRAIN	TRANSP_SXB_NIC_TRAIN	0	2
TRANSP_SXB_PAR_AMB_ACT	0	TRANSP_SXB_PAR_AMB_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_AMB_ACT	1	TRANSP_SXB_PAR_AMB_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2
TRANSP_SXB_PAR_BLD_ACT	0	TRANSP_SXB_PAR_BLD_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_BLD_ACT	1	TRANSP_SXB_PAR_BLD_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2
TRANSP_SXB_PAR_BRO_ACT	0	TRANSP_SXB_PAR_BRO_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_BRO_ACT	1	TRANSP_SXB_PAR_BRO_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2

Record: 35 of 50

With transportation modes you can define the different ways of transporting along a delivery arc. In this example you can transport by truck or train, corresponding to modes 0 and 1 for product delivery along any particular arc. Transporting by truck is half the variable cost of transporting by train.



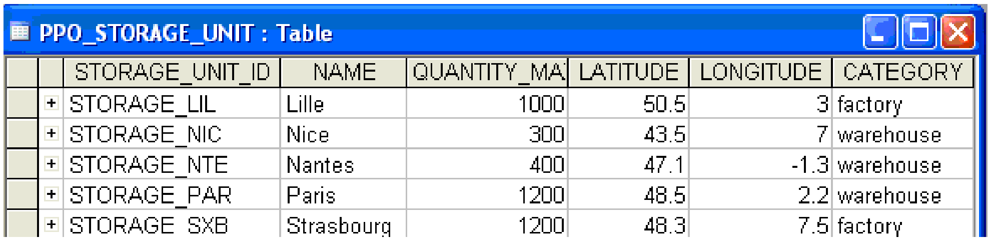
PPO_MODE : Table

ACTIVITY_ID	MODE	NAME	RESOURCE_ID	VARIABLE_PROCESSING_TIME	VARIABLE_COST
FAB_LIL_BLD_ACTIVITY	0	FAB_LIL_BLD	FAB_LIL	0.021	1
FAB_LIL_BRO_ACTIVITY	0	FAB_LIL_BRO	FAB_LIL	0.019	1
FAB_SXB_AMB_ACTIVITY	0	FAB_SXB_AMB	FAB_SXB	0.025	1
FAB_SXB_BLD_ACTIVITY	0	FAB_SXB_BLD	FAB_SXB	0.023	1
FAB_SXB_BRO_ACTIVITY	0	FAB_SXB_BRO	FAB_SXB	0.022	1
TRANSP_LIL_NTE_BRO_ACTI	0	TRANSP_LIL_NTE_BRO_TRUCK	TRANSP_LIL_NTE_TRUCK	0	1
TRANSP_LIL_NTE_BRO_ACTI	1	TRANSP_LIL_NTE_BRO_TRAIN	TRANSP_LIL_NTE_TRAIN	0	2
TRANSP_LIL_PAR_BLD_ACTI	0	TRANSP_LIL_PAR_BLD_TRUCK	TRANSP_LIL_PAR_TRUCK	0	1
TRANSP_LIL_PAR_BLD_ACTI	1	TRANSP_LIL_PAR_BLD_TRAIN	TRANSP_LIL_PAR_TRAIN	0	2
TRANSP_LIL_PAR_BRO_ACTI	0	TRANSP_LIL_PAR_BRO_TRUCK	TRANSP_LIL_PAR_TRUCK	0	1
TRANSP_LIL_PAR_BRO_ACTI	1	TRANSP_LIL_PAR_BRO_TRAIN	TRANSP_LIL_PAR_TRAIN	0	2
TRANSP_PAR_LIL_AMB_ACTI	0	TRANSP_PAR_LIL_AMB_TRUCK	TRANSP_PAR_LIL_TRUCK	0	1
TRANSP_PAR_LIL_AMB_ACTI	1	TRANSP_PAR_LIL_AMB_TRAIN	TRANSP_PAR_LIL_TRAIN	0	2
TRANSP_PAR_NIC_AMB_ACT	0	TRANSP_PAR_NIC_AMB_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_AMB_ACT	1	TRANSP_PAR_NIC_AMB_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NIC_BLD_ACTI	0	TRANSP_PAR_NIC_BLD_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_BLD_ACTI	1	TRANSP_PAR_NIC_BLD_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NIC_BRO_ACTI	0	TRANSP_PAR_NIC_BRO_TRUCK	TRANSP_PAR_NIC_TRUCK	0	1
TRANSP_PAR_NIC_BRO_ACTI	1	TRANSP_PAR_NIC_BRO_TRAIN	TRANSP_PAR_NIC_TRAIN	0	2
TRANSP_PAR_NTE_AMB_ACT	0	TRANSP_PAR_NTE_AMB_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_AMB_ACT	1	TRANSP_PAR_NTE_AMB_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_PAR_NTE_BLD_ACT	0	TRANSP_PAR_NTE_BLD_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_BLD_ACT	1	TRANSP_PAR_NTE_BLD_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_PAR_NTE_BRO_ACT	0	TRANSP_PAR_NTE_BRO_TRUCK	TRANSP_PAR_NTE_TRUCK	0	1
TRANSP_PAR_NTE_BRO_ACT	1	TRANSP_PAR_NTE_BRO_TRAIN	TRANSP_PAR_NTE_TRAIN	0	2
TRANSP_SXB_NIC_AMB_ACTI	0	TRANSP_SXB_NIC_AMB_TRUCK	TRANSP_SXB_NIC_TRUCK	0	1
TRANSP_SXB_NIC_AMB_ACTI	1	TRANSP_SXB_NIC_AMB_TRAIN	TRANSP_SXB_NIC_TRAIN	0	2
TRANSP_SXB_NIC_BRO_ACTI	0	TRANSP_SXB_NIC_BRO_TRUCK	TRANSP_SXB_NIC_TRUCK	0	1
TRANSP_SXB_NIC_BRO_ACTI	1	TRANSP_SXB_NIC_BRO_TRAIN	TRANSP_SXB_NIC_TRAIN	0	2
TRANSP_SXB_PAR_AMB_ACT	0	TRANSP_SXB_PAR_AMB_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_AMB_ACT	1	TRANSP_SXB_PAR_AMB_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2
TRANSP_SXB_PAR_BLD_ACT	0	TRANSP_SXB_PAR_BLD_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_BLD_ACT	1	TRANSP_SXB_PAR_BLD_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2
TRANSP_SXB_PAR_BRO_ACT	0	TRANSP_SXB_PAR_BRO_TRUCK	TRANSP_SXB_PAR_TRUCK	0	1
TRANSP_SXB_PAR_BRO_ACT	1	TRANSP_SXB_PAR_BRO_TRAIN	TRANSP_SXB_PAR_TRAIN	0	2

Record: 35 of 50

Storage units

Storage units model sites within the distribution network: Plants for production sites, warehouses for storage, and customers for the marketplace. Some sites can be merged: Warehouses can be merged with the corresponding plant, and the customer sites can be merged to a plant or warehouse. You can define the storage capacity of the site with the QUANTITY_MAX field. You can check this later in the results with the Warehouse Summary view, and you can define latitude and longitude to visualize the sites in the Distribution Planning view.

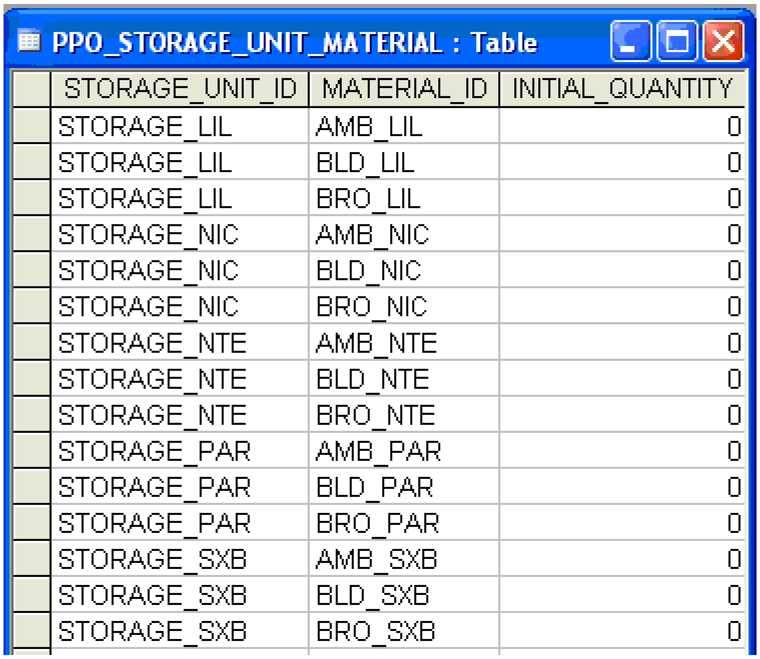


PPO_STORAGE_UNIT : Table

STORAGE_UNIT_ID	NAME	QUANTITY_MA	LATITUDE	LONGITUDE	CATEGORY
+ STORAGE_LIL	Lille	1000	50.5	3	factory
+ STORAGE_NIC	Nice	300	43.5	7	warehouse
+ STORAGE_NTE	Nantes	400	47.1	-1.3	warehouse
+ STORAGE_PAR	Paris	1200	48.5	2.2	warehouse
+ STORAGE_SXB	Strasbourg	1200	48.3	7.5	factory

Storage unit material

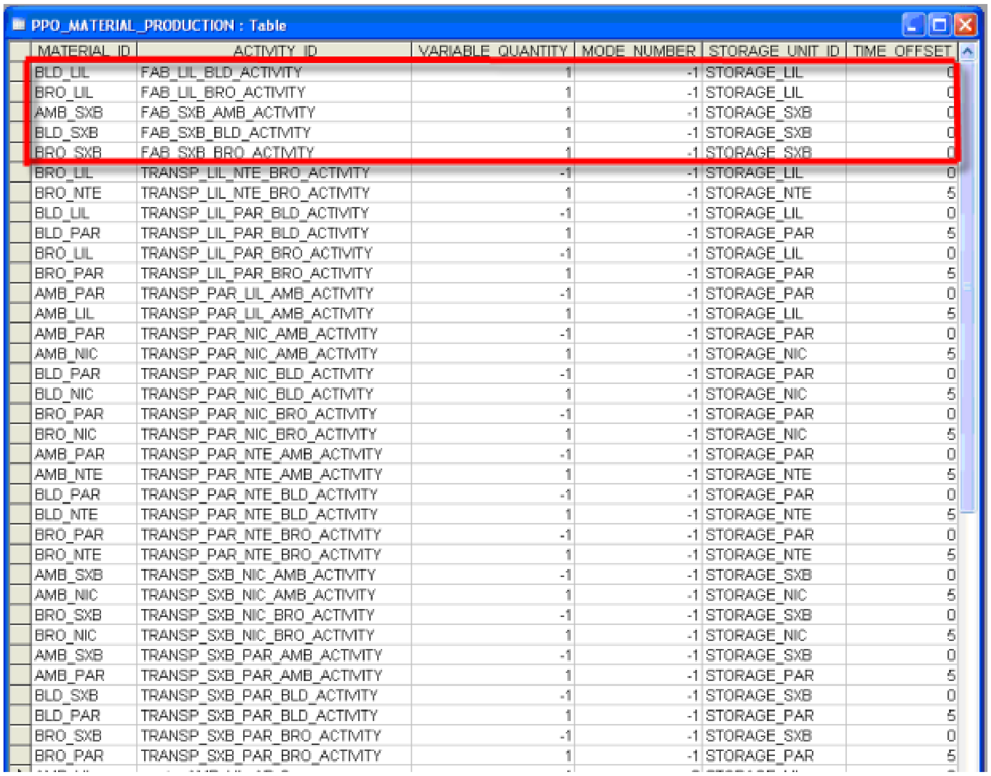
For each storage unit (site) you can define the initial quantity of each material present at the start of the planning horizon.



STORAGE_UNIT_ID	MATERIAL_ID	INITIAL_QUANTITY
STORAGE_LIL	AMB_LIL	0
STORAGE_LIL	BLD_LIL	0
STORAGE_LIL	BRO_LIL	0
STORAGE_NIC	AMB_NIC	0
STORAGE_NIC	BLD_NIC	0
STORAGE_NIC	BRO_NIC	0
STORAGE_NTE	AMB_NTE	0
STORAGE_NTE	BLD_NTE	0
STORAGE_NTE	BRO_NTE	0
STORAGE_PAR	AMB_PAR	0
STORAGE_PAR	BLD_PAR	0
STORAGE_PAR	BRO_PAR	0
STORAGE_SXB	AMB_SXB	0
STORAGE_SXB	BLD_SXB	0
STORAGE_SXB	BRO_SXB	0

Material production

Material production on production recipes creates material on the storage unit of the associated plant.

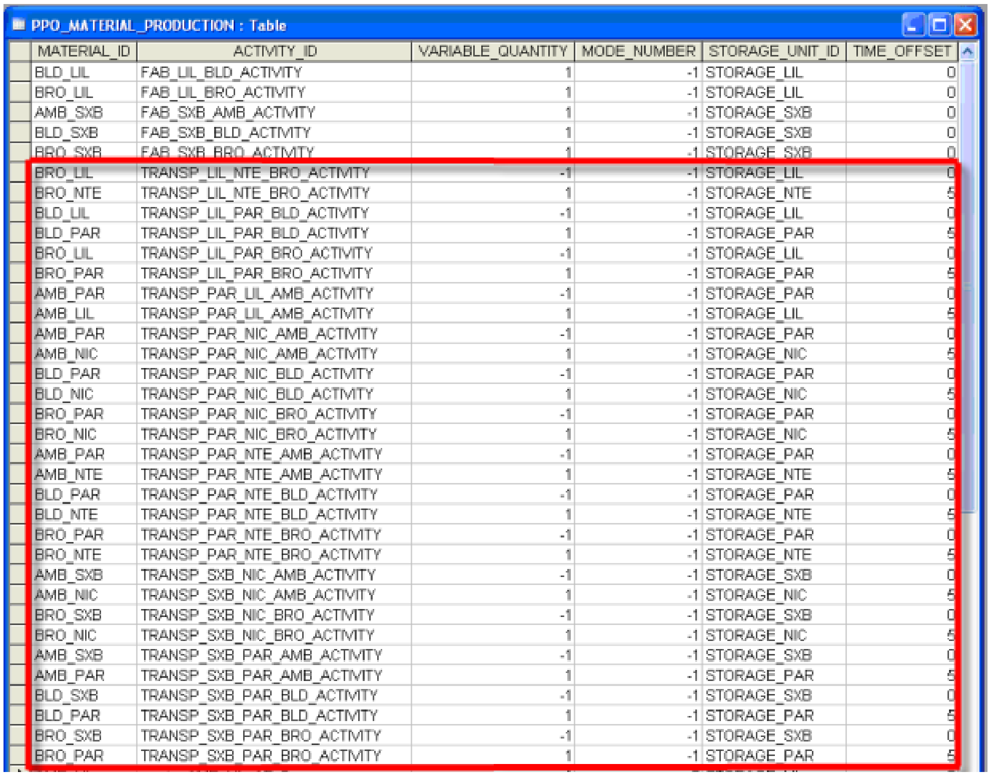


PPO_MATERIAL_PRODUCTION : Table

MATERIAL_ID	ACTIVITY_ID	VARIABLE_QUANTITY	MODE_NUMBER	STORAGE_UNIT_ID	TIME_OFFSET
BLD_LIL	FAB_LIL_BLD_ACTIVITY	1	-1	STORAGE_LIL	0
BRO_LIL	FAB_LIL_BRO_ACTIVITY	1	-1	STORAGE_LIL	0
AMB_SXB	FAB_SXB_AMB_ACTIVITY	1	-1	STORAGE_SXB	0
BLD_SXB	FAB_SXB_BLD_ACTIVITY	1	-1	STORAGE_SXB	0
BRO_SXB	FAB_SXB_BRO_ACTIVITY	1	-1	STORAGE_SXB	0
BRO_LIL	TRANSP_LIL_NTE_BRO_ACTIVITY	-1	-1	STORAGE_LIL	0
BRO_NTE	TRANSP_LIL_NTE_BRO_ACTIVITY	1	-1	STORAGE_NTE	5
BLD_LIL	TRANSP_LIL_PAR_BLD_ACTIVITY	-1	-1	STORAGE_LIL	0
BLD_PAR	TRANSP_LIL_PAR_BLD_ACTIVITY	1	-1	STORAGE_PAR	5
BRO_LIL	TRANSP_LIL_PAR_BRO_ACTIVITY	-1	-1	STORAGE_LIL	0
BRO_PAR	TRANSP_LIL_PAR_BRO_ACTIVITY	1	-1	STORAGE_PAR	5
AMB_PAR	TRANSP_PAR_LIL_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_LIL	TRANSP_PAR_LIL_AMB_ACTIVITY	1	-1	STORAGE_LIL	5
AMB_PAR	TRANSP_PAR_NIC_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_NIC	TRANSP_PAR_NIC_AMB_ACTIVITY	1	-1	STORAGE_NIC	5
BLD_PAR	TRANSP_PAR_NIC_BLD_ACTIVITY	-1	-1	STORAGE_PAR	0
BLD_NIC	TRANSP_PAR_NIC_BLD_ACTIVITY	1	-1	STORAGE_NIC	5
BRO_PAR	TRANSP_PAR_NIC_BRO_ACTIVITY	-1	-1	STORAGE_PAR	0
BRO_NIC	TRANSP_PAR_NIC_BRO_ACTIVITY	1	-1	STORAGE_NIC	5
AMB_PAR	TRANSP_PAR_NTE_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_NTE	TRANSP_PAR_NTE_AMB_ACTIVITY	1	-1	STORAGE_NTE	5
BLD_PAR	TRANSP_PAR_NTE_BLD_ACTIVITY	-1	-1	STORAGE_PAR	0
BLD_NTE	TRANSP_PAR_NTE_BLD_ACTIVITY	1	-1	STORAGE_NTE	5
BRO_PAR	TRANSP_PAR_NTE_BRO_ACTIVITY	-1	-1	STORAGE_PAR	0
BRO_NTE	TRANSP_PAR_NTE_BRO_ACTIVITY	1	-1	STORAGE_NTE	5
AMB_SXB	TRANSP_SXB_NIC_AMB_ACTIVITY	-1	-1	STORAGE_SXB	0
AMB_NIC	TRANSP_SXB_NIC_AMB_ACTIVITY	1	-1	STORAGE_NIC	5
BRO_SXB	TRANSP_SXB_NIC_BRO_ACTIVITY	-1	-1	STORAGE_SXB	0
BRO_NIC	TRANSP_SXB_NIC_BRO_ACTIVITY	1	-1	STORAGE_NIC	5
AMB_SXB	TRANSP_SXB_PAR_AMB_ACTIVITY	-1	-1	STORAGE_SXB	0
AMB_PAR	TRANSP_SXB_PAR_AMB_ACTIVITY	1	-1	STORAGE_PAR	5
BLD_SXB	TRANSP_SXB_PAR_BLD_ACTIVITY	-1	-1	STORAGE_SXB	0
BLD_PAR	TRANSP_SXB_PAR_BLD_ACTIVITY	1	-1	STORAGE_PAR	5
BRO_SXB	TRANSP_SXB_PAR_BRO_ACTIVITY	-1	-1	STORAGE_SXB	0
BRO_PAR	TRANSP_SXB_PAR_BRO_ACTIVITY	1	-1	STORAGE_PAR	5

For transportation recipes, material production:

- Consumes material on the storage unit of the incoming site;
- Produces *another* material on the storage unit of the outgoing site.

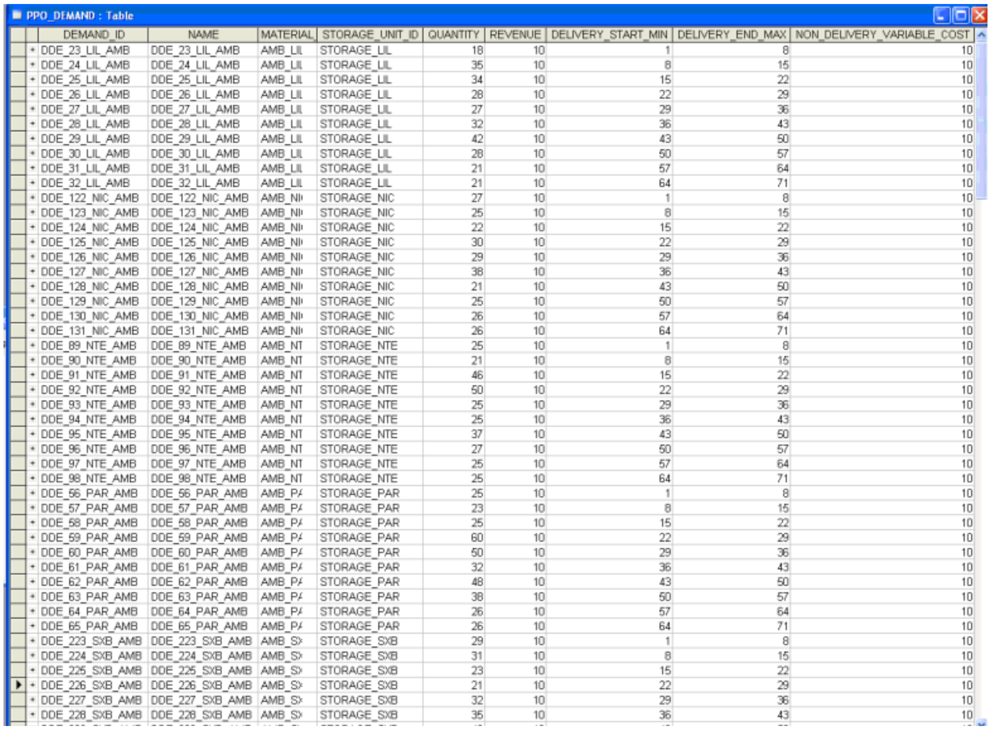


PPO_MATERIAL_PRODUCTION : Table

MATERIAL_ID	ACTIVITY_ID	VARIABLE_QUANTITY	MODE_NUMBER	STORAGE_UNIT_ID	TIME_OFFSET
BLD_LIL	FAB_LIL_BLD_ACTIVITY	1	-1	STORAGE_LIL	0
BRO_LIL	FAB_LIL_BRO_ACTIVITY	1	-1	STORAGE_LIL	0
AMB_SXB	FAB_SXB_AMB_ACTIVITY	1	-1	STORAGE_SXB	0
BLD_SXB	FAB_SXB_BLD_ACTIVITY	1	-1	STORAGE_SXB	0
BRO_SXB	FAB_SXB_BRO_ACTIVITY	1	-1	STORAGE_SXB	0
BRO_LIL	TRANSP_LIL_NTE_BRO_ACTIVITY	-1	-1	STORAGE_LIL	0
BRO_NTE	TRANSP_LIL_NTE_BRO_ACTIVITY	1	-1	STORAGE_NTE	5
BLD_LIL	TRANSP_LIL_PAR_BLD_ACTIVITY	-1	-1	STORAGE_LIL	0
BLD_PAR	TRANSP_LIL_PAR_BLD_ACTIVITY	1	-1	STORAGE_PAR	5
BRO_LIL	TRANSP_LIL_PAR_BRO_ACTIVITY	-1	-1	STORAGE_LIL	0
BRO_PAR	TRANSP_LIL_PAR_BRO_ACTIVITY	1	-1	STORAGE_PAR	5
AMB_PAR	TRANSP_PAR_LIL_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_LIL	TRANSP_PAR_LIL_AMB_ACTIVITY	1	-1	STORAGE_LIL	5
AMB_PAR	TRANSP_PAR_NIC_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_NIC	TRANSP_PAR_NIC_AMB_ACTIVITY	1	-1	STORAGE_NIC	5
BLD_PAR	TRANSP_PAR_NIC_BLD_ACTIVITY	-1	-1	STORAGE_PAR	0
BLD_NIC	TRANSP_PAR_NIC_BLD_ACTIVITY	1	-1	STORAGE_NIC	5
BRO_PAR	TRANSP_PAR_NIC_BRO_ACTIVITY	-1	-1	STORAGE_PAR	0
BRO_NIC	TRANSP_PAR_NIC_BRO_ACTIVITY	1	-1	STORAGE_NIC	5
AMB_PAR	TRANSP_PAR_NTE_AMB_ACTIVITY	-1	-1	STORAGE_PAR	0
AMB_NTE	TRANSP_PAR_NTE_AMB_ACTIVITY	1	-1	STORAGE_NTE	5
BLD_PAR	TRANSP_PAR_NTE_BLD_ACTIVITY	-1	-1	STORAGE_PAR	0
BLD_NTE	TRANSP_PAR_NTE_BLD_ACTIVITY	1	-1	STORAGE_NTE	5
BRO_PAR	TRANSP_PAR_NTE_BRO_ACTIVITY	-1	-1	STORAGE_PAR	0
BRO_NTE	TRANSP_PAR_NTE_BRO_ACTIVITY	1	-1	STORAGE_NTE	5
AMB_SXB	TRANSP_SXB_NIC_AMB_ACTIVITY	-1	-1	STORAGE_SXB	0
AMB_NIC	TRANSP_SXB_NIC_AMB_ACTIVITY	1	-1	STORAGE_NIC	5
BRO_SXB	TRANSP_SXB_NIC_BRO_ACTIVITY	-1	-1	STORAGE_SXB	0
BRO_NIC	TRANSP_SXB_NIC_BRO_ACTIVITY	1	-1	STORAGE_NIC	5
AMB_SXB	TRANSP_SXB_PAR_AMB_ACTIVITY	-1	-1	STORAGE_SXB	0
AMB_PAR	TRANSP_SXB_PAR_AMB_ACTIVITY	1	-1	STORAGE_PAR	5
BLD_SXB	TRANSP_SXB_PAR_BLD_ACTIVITY	-1	-1	STORAGE_SXB	0
BLD_PAR	TRANSP_SXB_PAR_BLD_ACTIVITY	1	-1	STORAGE_PAR	5
BRO_SXB	TRANSP_SXB_PAR_BRO_ACTIVITY	-1	-1	STORAGE_SXB	0
BRO_PAR	TRANSP_SXB_PAR_BRO_ACTIVITY	1	-1	STORAGE_PAR	5

Demands

It is preferable to merge the demands for a product in a marketplace and thereby have only value for each paired *Product*:*Site*. But if you desire you can maintain some details and allow the merger to occur automatically in the planning calculation.

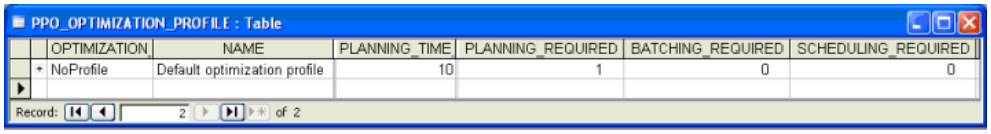


PPO_DEMAND : Table

DEMAND_ID	NAME	MATERIAL	STORAGE_UNIT_ID	QUANTITY	REVENUE	DELIVERY_START_MIN	DELIVERY_END_MAX	NON_DELIVERY_VARIABLE_COST
DDE_23_LIL_AMB	DDE_23_LIL_AMB	AMB_LIL	STORAGE_LIL	18	10	1	8	10
DDE_24_LIL_AMB	DDE_24_LIL_AMB	AMB_LIL	STORAGE_LIL	35	10	8	15	10
DDE_25_LIL_AMB	DDE_25_LIL_AMB	AMB_LIL	STORAGE_LIL	34	10	15	22	10
DDE_26_LIL_AMB	DDE_26_LIL_AMB	AMB_LIL	STORAGE_LIL	28	10	22	29	10
DDE_27_LIL_AMB	DDE_27_LIL_AMB	AMB_LIL	STORAGE_LIL	27	10	29	36	10
DDE_28_LIL_AMB	DDE_28_LIL_AMB	AMB_LIL	STORAGE_LIL	32	10	36	43	10
DDE_29_LIL_AMB	DDE_29_LIL_AMB	AMB_LIL	STORAGE_LIL	42	10	43	50	10
DDE_30_LIL_AMB	DDE_30_LIL_AMB	AMB_LIL	STORAGE_LIL	28	10	50	57	10
DDE_31_LIL_AMB	DDE_31_LIL_AMB	AMB_LIL	STORAGE_LIL	21	10	57	64	10
DDE_32_LIL_AMB	DDE_32_LIL_AMB	AMB_LIL	STORAGE_LIL	21	10	64	71	10
DDE_122_NIC_AMB	DDE_122_NIC_AMB	AMB_NI	STORAGE_NIC	27	10	1	8	10
DDE_123_NIC_AMB	DDE_123_NIC_AMB	AMB_NI	STORAGE_NIC	25	10	8	15	10
DDE_124_NIC_AMB	DDE_124_NIC_AMB	AMB_NI	STORAGE_NIC	22	10	15	22	10
DDE_125_NIC_AMB	DDE_125_NIC_AMB	AMB_NI	STORAGE_NIC	30	10	22	29	10
DDE_126_NIC_AMB	DDE_126_NIC_AMB	AMB_NI	STORAGE_NIC	29	10	29	36	10
DDE_127_NIC_AMB	DDE_127_NIC_AMB	AMB_NI	STORAGE_NIC	38	10	36	43	10
DDE_128_NIC_AMB	DDE_128_NIC_AMB	AMB_NI	STORAGE_NIC	21	10	43	50	10
DDE_129_NIC_AMB	DDE_129_NIC_AMB	AMB_NI	STORAGE_NIC	25	10	50	57	10
DDE_130_NIC_AMB	DDE_130_NIC_AMB	AMB_NI	STORAGE_NIC	26	10	57	64	10
DDE_131_NIC_AMB	DDE_131_NIC_AMB	AMB_NI	STORAGE_NIC	26	10	64	71	10
DDE_89_NTE_AMB	DDE_89_NTE_AMB	AMB_NT	STORAGE_NTE	25	10	1	8	10
DDE_90_NTE_AMB	DDE_90_NTE_AMB	AMB_NT	STORAGE_NTE	21	10	8	15	10
DDE_91_NTE_AMB	DDE_91_NTE_AMB	AMB_NT	STORAGE_NTE	46	10	15	22	10
DDE_92_NTE_AMB	DDE_92_NTE_AMB	AMB_NT	STORAGE_NTE	50	10	22	29	10
DDE_93_NTE_AMB	DDE_93_NTE_AMB	AMB_NT	STORAGE_NTE	25	10	29	36	10
DDE_94_NTE_AMB	DDE_94_NTE_AMB	AMB_NT	STORAGE_NTE	25	10	36	43	10
DDE_95_NTE_AMB	DDE_95_NTE_AMB	AMB_NT	STORAGE_NTE	37	10	43	50	10
DDE_96_NTE_AMB	DDE_96_NTE_AMB	AMB_NT	STORAGE_NTE	27	10	50	57	10
DDE_97_NTE_AMB	DDE_97_NTE_AMB	AMB_NT	STORAGE_NTE	25	10	57	64	10
DDE_98_NTE_AMB	DDE_98_NTE_AMB	AMB_NT	STORAGE_NTE	25	10	64	71	10
DDE_56_PAR_AMB	DDE_56_PAR_AMB	AMB_PA	STORAGE_PAR	25	10	1	8	10
DDE_57_PAR_AMB	DDE_57_PAR_AMB	AMB_PA	STORAGE_PAR	23	10	8	15	10
DDE_58_PAR_AMB	DDE_58_PAR_AMB	AMB_PA	STORAGE_PAR	25	10	15	22	10
DDE_59_PAR_AMB	DDE_59_PAR_AMB	AMB_PA	STORAGE_PAR	60	10	22	29	10
DDE_60_PAR_AMB	DDE_60_PAR_AMB	AMB_PA	STORAGE_PAR	50	10	29	36	10
DDE_61_PAR_AMB	DDE_61_PAR_AMB	AMB_PA	STORAGE_PAR	32	10	36	43	10
DDE_62_PAR_AMB	DDE_62_PAR_AMB	AMB_PA	STORAGE_PAR	48	10	43	50	10
DDE_63_PAR_AMB	DDE_63_PAR_AMB	AMB_PA	STORAGE_PAR	38	10	50	57	10
DDE_64_PAR_AMB	DDE_64_PAR_AMB	AMB_PA	STORAGE_PAR	26	10	57	64	10
DDE_65_PAR_AMB	DDE_65_PAR_AMB	AMB_PA	STORAGE_PAR	26	10	64	71	10
DDE_223_SXB_AMB	DDE_223_SXB_AMB	AMB_SX	STORAGE_SXB	29	10	1	8	10
DDE_224_SXB_AMB	DDE_224_SXB_AMB	AMB_SX	STORAGE_SXB	31	10	8	15	10
DDE_225_SXB_AMB	DDE_225_SXB_AMB	AMB_SX	STORAGE_SXB	23	10	15	22	10
DDE_226_SXB_AMB	DDE_226_SXB_AMB	AMB_SX	STORAGE_SXB	21	10	22	29	10
DDE_227_SXB_AMB	DDE_227_SXB_AMB	AMB_SX	STORAGE_SXB	32	10	29	36	10
DDE_228_SXB_AMB	DDE_228_SXB_AMB	AMB_SX	STORAGE_SXB	35	10	36	43	10

Optimization profile

Only the planning module is required in this planning example.



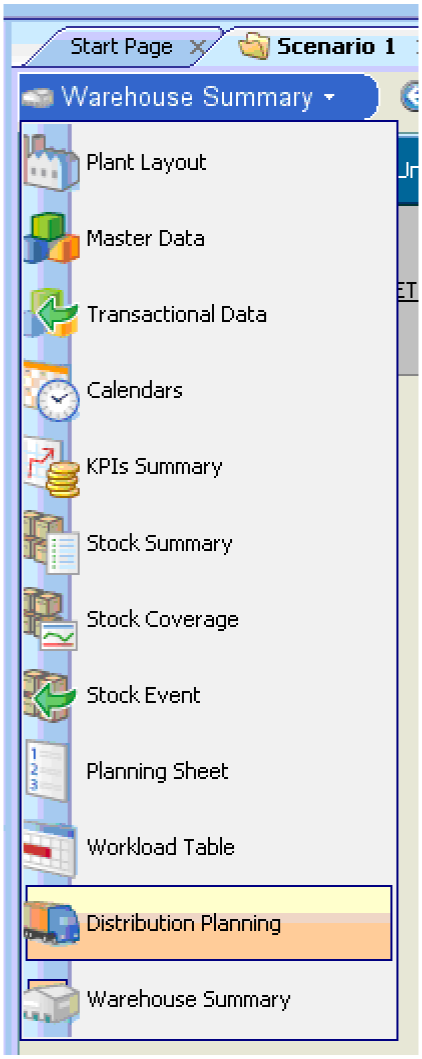
PPO_OPTIMIZATION_PROFILE : Table

OPTIMIZATION	NAME	PLANNING_TIME	PLANNING_REQUIRED	BATCHING_REQUIRED	SCHEDULING_REQUIRED
NoProfile	Default optimization profile	10	1	0	0

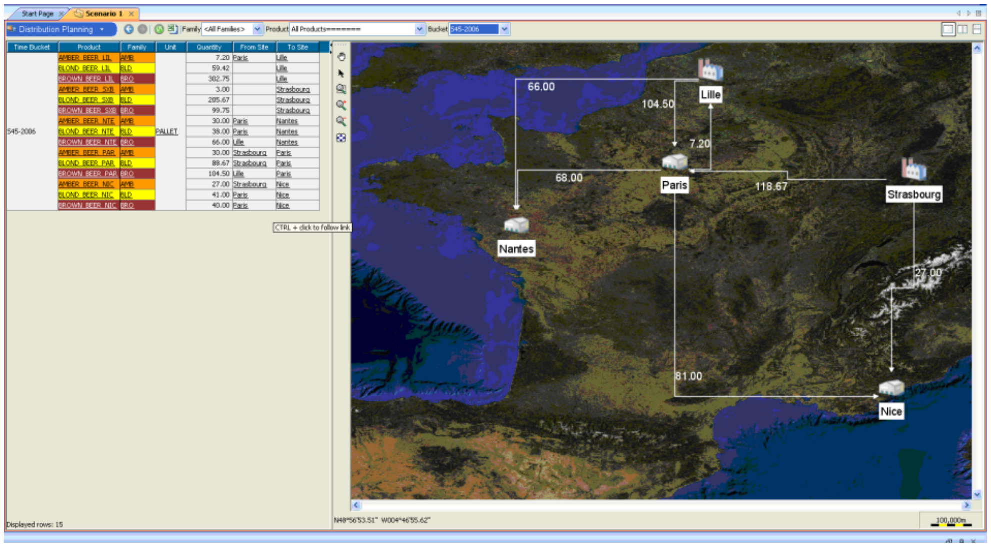
Record: 2 of 2

Viewing the results

If the PPO_SETTING map file has been defined (see *Map graphic for the Planning View* in *Building a distribution plan model*) then two views are available in the GUI that reveal the results of the planning: **Distribution Planning** and **Warehouse Summary**.

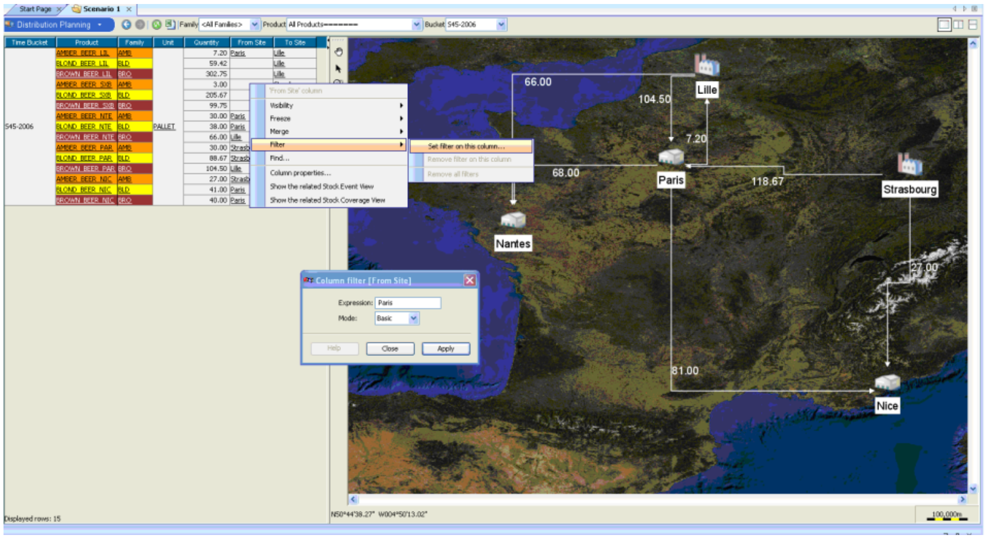


The **Distribution Planning** view displays the flow of material between locations in tabular form on the left and graphically on the map on the right. You can navigate via families, materials or buckets.



Time Bucket	Product	Family	Unit	Quantity	From Site	To Site
545-2006	AMBER_BEER_LIL	AMB	PALLET	7.20	Paris	Lille
	BLOND_BEER_LIL	BLD		59.42		Lille
	BROWN_BEER_LIL	BRO		302.75		Lille
	AMBER_BEER_SXB	AMB		3.00		Strasbourg
	BLOND_BEER_SXB	BLD		205.67		Strasbourg
	BROWN_BEER_SXB	BRO		99.75		Strasbourg
	AMBER_BEER_NTE	AMB		30.00	Paris	Nantes
	BLOND_BEER_NTE	BLD		38.00	Paris	Nantes
	BROWN_BEER_NTE	BRO		66.00	Lille	Nantes
	AMBER_BEER_PAR	AMB		30.00	Strasbourg	Paris
	BLOND_BEER_PAR	BLD		88.67	Strasbourg	Paris
	BROWN_BEER_PAR	BRO		104.50	Lille	Paris
	AMBER_BEER_NIC	AMB		27.00	Strasbourg	Nice
	BLOND_BEER_NIC	BLD		41.00	Paris	Nice
	BROWN_BEER_NIC	BRO		40.00	Paris	Nice

Use filters to selectively display parts of a multi-plant solution. Activate filters on the left tabular view by using the right-click contextual menu; the filtered column is highlighted (underlined). You can remove and modify filters from the same menu. In the example below, a filter is set so that only arcs coming out of Paris will be displayed.



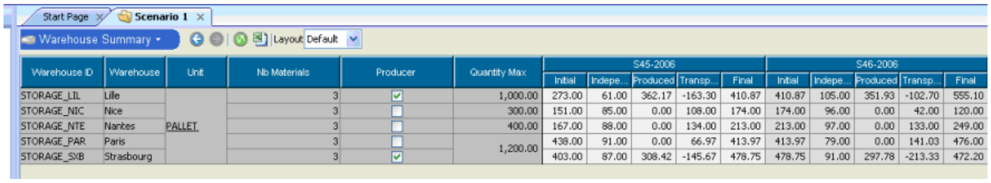
Time Bucket	Product	Family	Unit	Quantity	From Site	To Site
545-2006	AMBER_BEER_LIL	AMB	PALLET	7.20	Paris	Lille
	BLOND_BEER_LIL	BLD		59.42		Lille
	BROWN_BEER_LIL	BRO		302.75		Lille
	AMBER_BEER_SXB	AMB		3.00		
	BLOND_BEER_SXB	BLD		205.67		
	BROWN_BEER_SXB	BRO		99.75		
	AMBER_BEER_NTE	AMB		30.00	Paris	
	BLOND_BEER_NTE	BLD		38.00	Paris	
	BROWN_BEER_NTE	BRO		66.00	Lille	
	AMBER_BEER_PAR	AMB		30.00	Strasb	
	BLOND_BEER_PAR	BLD		88.67	Strasb	
	BROWN_BEER_PAR	BRO		104.50	Lille	
	AMBER_BEER_NIC	AMB		27.00	Strasb	
	BLOND_BEER_NIC	BLD		41.00	Paris	
	BROWN_BEER_NIC	BRO		40.00	Paris	

Column filter [From Site]

Expression: Paris

Mode: Basic

In the Warehouse Summary view you can see the storage unit sites and their properties. A checkbox indicates if the site is a production plant; maximum storage is displayed (**Quantity Max**); and for each bucket you have details regarding initial and final stock, demands, and produced or transported quantities (from that site).



Warehouse ID	Warehouse	Unit	Nb Materials	Producer	Quantity Max	S45-2006					S46-2006				
						Initial	Indepe...	Produced	Transp...	Final	Initial	Indepe...	Produced	Transp...	Final
STORAGE_LIL	Lille		3	✓	1,000.00	273.00	61.00	362.17	-163.30	410.87	410.87	105.00	351.93	-102.70	555.10
STORAGE_NIC	Nice		3	☐	300.00	151.00	85.00	0.00	108.00	174.00	174.00	96.00	0.00	42.00	120.00
STORAGE_NTE	Nantes	PALLET	3	☐	400.00	167.00	88.00	0.00	134.00	213.00	213.00	97.00	0.00	133.00	249.00
STORAGE_PAR	Paris		3	☐	1,200.00	438.00	91.00	0.00	66.97	413.97	413.97	79.00	0.00	141.03	476.00
STORAGE_SXB	Strasbourg		3	✓		403.00	87.00	308.42	-145.67	478.75	478.75	91.00	297.78	-213.33	472.20

Examples and tutorials

This section includes examples and tutorials showing how to build a data model in Plant PowerOps. The examples range from basic to advanced, and include use of database files as well as the C++ and Java™ APIs. The first example models a basic problem using spreadsheet (csv) and database (mdb) files. The C++ and Java sections model a problem using calendars. The final example uses the Java API to model yogurt production at a fresh dairy plant, including tank modeling and activity cleanups.

In this section

Modeling a simple problem: A “bottleneck” resource
Describes the entire process of using PPO to model and solve a simple manufacturing problem.

Using the PPO API for C++ to model and solve
The first section describes how to use C++ to solve a problem model built with csv database files. The second section describes how to use the C++ API to model and solve a scheduling problem that uses calendars. You may also wish to refer to *Using PPO with Microsoft products*.

Using the PPO API for Java to model and solve
The first section describes how to use Java™ to solve a problem model built with csv database files. The second section describes how to use the Java API to model and solve a scheduling problem that uses calendars.

Modeling a dairy plant with PPO Java API
This section models the manufacturing process of a hypothetical dairy plant, using the PPO Java™ API. First the actual manufacturing process is examined; from this the appropriate PPO model is defined and implemented in the Java API. Only some of the more interesting aspects of modeling this problem are described here, but the complete code example is located at `<PPOInstallDirectory>\examples\src\dairyplant.java`.

Modeling a simple problem: A “bottleneck” resource

Describes the entire process of using PPO to model and solve a simple manufacturing problem.

In this section

Overview
Introduces the problem to be solved.

Describe the problem
Details all information necessary to model the problem.

An overview of the PPO data model
Introduces the basic elements of the PPO data model necessary to model this manufacturing problem.

A quick tutorial on csv and mdb usage in PPO
A basic introduction to using csv and mdb files to contain problem data.

Model the problem
Describes the actual process of coding the problem data into a model file that PPO can read and optimize.

Solving
Describes how to use PPO to solve the problem data previously encoded.

View and study the plan
Shows you how to read and comprehend the results from an optimization in PPO.

Review

A review of this lesson.

Overview

In this first lesson, you will learn how to:

- ♦ Analyze and describe a manufacturing problem in terms of Plant PowerOps (PPO) resources, activities, demands, and other data model objects;
- ♦ Create a model of the problem in a series of data tables, using spreadsheet (comma-separated values) files and database (Microsoft® Access® database) files;
- ♦ Solve the problem by loading the csv or mdb file into the PPO GUI (Graphical User Interface);
- ♦ View the problem data and solution in the PPO GUI.

The manufacturing problem we look at here involves a factory that makes wooden cradles. In particular, we're going to examine use of a crucial resource in the factory: the dryer which dries cradles after they have been varnished.

This example is very simple in order to introduce you to many elements of PPO, including the data model, modeling with data tables, and use of the PPO GUI. Of particular note, this is a pure scheduling problem with all demands due for a given day, and therefore does not involve production planning. More complex examples are available for your study in the standard delivery, at <*PPOInstallDirectory*>`\examples\data\course` and in the subfolder `mdb`.

As noted above, the first thing to do is analyze and describe the problem.

Describe the problem

Details all information necessary to model the problem.

In this section

Overview
Describes the particulars of the cradle manufacturing process.

Objective and costs
Describes the objective and costs of solving the problem.

Before creating the model
Discusses the implications of choosing a modeling methodology.

Overview

The crucial step in this problem uses the dryer machine to dry cradles. It represents a "bottleneck" in the manufacturing process, and that’s why it’s interesting; maximizing efficiency with this resource could speed production and save money.

The dryer machine is a true bottleneck step because:

- ♦ All cradles must be dried after varnishing;
- ♦ This factory has only one dryer, and the dryer can accommodate only one cradle at a time;
- ♦ Since varnishing takes less time than drying, many cradles may be varnished and waiting for the dryer at the same time.

There are two types of wooden cradles, pine and teak. Pine cradles require the dryer resource for 70 minutes, and teak cradles require 90 minutes.

There are outstanding customer orders for ten cradles, three of which are pine and seven are teak. Since we are concerned only with the dryer in this first example, we assume that all cradles are already built, varnished and awaiting drying at the start of the problem.

To meet customer demands, the drying activities for the three pine cradles should be completed at time 160, time 680, and time 720. Likewise, the drying process for the seven teak cradles should be completed at times 80, 120, 280, 280, 560, 640, and 800. (Each time unit is a minute in this example; more information about time units follows.)

As you can quickly see, this example has potential for conflict. For example, the first teak cradle is supposed to be ready at time 80, yet teak cradles take longer than that (90 time units) to dry. Two teak cradles are supposed to be completed at time 280, yet only one cradle can be dried at a time; if the dryer is already running at full capacity at around time 190, this will also be a problem.

Objective and costs

The objective of this process is the same as for most businesses: Meet customer demands to obtain revenue, while simultaneously keeping internal costs to a minimum. In this example, that means that drying activities should finish as close as possible to the ideal due dates. This keeps customers happy, and reduces the costs associated with late or early completion.

However, resource constraints in a factory might mean that the ideal targets are not met (and in this example it’s clear that will indeed be the case). Late completion risks customer dissatisfaction and the possibility that they will choose another vendor in the future. Early completion might require that you pay storage costs before shipping to the customer. Plant PowerOps provides explicit methods to track these types of costs, regardless of whether the cost is actual (as when the cost produces a storage bill that must be paid), or whether the cost is potential (as in the potential loss of future business).

In this example, the cost for being late, or *tardy*, is greater than the earliness cost because we do not want to risk customer dissatisfaction. Also, the teak cradle customer is more important to the business than the pine cradle customer, because the teak customer places many more teak cradle orders than are received for pine cradles. So, the tardiness cost is appropriately greater for teak than it is for pine.

Additionally, costs can be either *variable* (that is, the cost is affected by the amount of time early or late) or *fixed*. A fixed cost is incurred if the activity is early or late, but there is no multiplication effect. For example, there might be a fixed cost to initially rent out or build a storage unit; and then a variable cost (extra wages, transportation, actual floor space used) that increases with the amount of time, or storage used or needed.

Taking these factors and past experience into account, this example assigns:

♦ An *earliness variable cost* of 1 and a *tardiness variable cost* of 3 to pine cradles, and

♦ An earliness variable cost of 1 and a tardiness variable cost of 7 to teak cradles.

Revenue and non-delivery costs

Each cradle that meets demand would (theoretically at least) bring in revenue. The "flip side" of revenue is nondelivery cost; the cost of failing to meet a demand at all. The nondelivery cost might simply represent the "absence" of revenue, or something more intangible such as customer dissatisfaction.

You could say that revenue and nondelivery costs together create an incentive to meet a demand. You could also say that if neither revenue nor nondelivery costs exist, then there is little or perhaps even no incentive to meet a demand. PPO actually requires that you apply a nondelivery cost to every demand that you have any expectation of being satisfied; revenue provides additional incentive to satisfy this demand.

In this example, we do not represent the revenue gained from delivered cradles, but we do model the nondelivery cost. Nondelivery of a product is far more severe a problem than simply being late. Being late with a delivery may leave you with a disgruntled customer; not delivering at all will likely lose you that customer completely.

So, we choose large penalties for nondelivery: 30000 for pine cradles, and 70000 for teak, reflecting the relative importance of the two customers.

Balancing nondelivery costs and tardiness costs

Nondelivery costs, earliness costs and tardiness costs must be defined consistently for PPO to make relevant compromises. Although you probably arrive at values for each of these costs independently, you should check and balance these values against each other before modeling the data.

In considering this example, there actually is no chance of having a nondelivery problem, as there is ample time and resource available. The only real issue in this example is how far behind schedule will the cradles be manufactured.

So, we should consider this question: How long past a due date does it still make sense to deliver a product?

A way to measure this is to divide the nondelivery variable cost by the tardiness variable cost, and see if it matches a reasonable amount. For pine cradles, we have 30000 nondelivery cost units divided by 3 tardiness cost units per time unit, leaving a result of 10000 time units. Teak cradles gives the same result, and as the time unit is a minute, this result is about a week for each type of cradle.

That seems reasonable; in this problem, it makes no sense to deliver more than a week late. If the delay becomes too long and increases the tardiness cost to very high levels, there comes a point (about one week) where it simply does not make sense to deliver. Our tardiness and nondelivery values are therefore in balance.

About time

The *date origin* for the problem represents the time *zero* for the problem. This is not necessarily the point in time at which PPO can first schedule an activity, however. That point is called the start minimum; in this example the start min is set to the date origin, and both are set to February 1, 2001, 00:00.

There is also a maximal end time for all scheduled activities in the problem.

The *time unit* for the problem is minutes (60 seconds). That is, start and end times are expressed in minutes from the date origin. Time *durations* are also expressed in minutes.

So the due dates for the pine cradles correspond to February 1, 2001, 02:40 (time 160), February 1, 2001, 11:20 (time 680), and February 1, 2001, 12:00 (time 720).

Before creating the model

We now have a lot of data with which to start building the problem model in data tables. We know the basic problem setup: the date origin and start min, the time unit, the activities that must be performed and their due dates. We know the resource (the dryer) used to perform the activities, its capacity, and the cost penalties associated with failing to meet the due dates. There are a few more things to consider, however.

Choosing a modeling methodology

This example is a simple one, presented just to get you introduced to PPO. However, in your real working life, you no doubt have a far more complicated data set, perhaps with thousands of interrelated, sequential activities to perform on a hundred resources. Some of the resources could be used in various ways, for different processes, on different materials. There might be intermediate materials that are transformed into other materials, and these intermediates may have shelf lives or storage restrictions.

So although you could model the problem described here in a very simple fashion, keep in mind that ultimately you want to use a design methodology that supports much greater complexity. So, as time passes, you can upgrade the model easily, to be more representative of the complexity of the process. The next section, *An overview of the PPO data model*, explains the modeling methodology used to enable such flexibility.

An overview of the PPO data model

Introduces the basic elements of the PPO data model necessary to model this manufacturing problem.

In this section

Model the process
Describes the goal of modeling the process rather than the explicit manufacturing activities.

Materials
Describes what a material is in the data model and the current manufacturing problem.

Demands
Describes what a *demand* is in the data model and the current manufacturing problem.

Activities
Describes what activities and *activity prototypes* are in the data model and the current manufacturing problem.

Resources
Describes what resources are in the data model and the current manufacturing problem.

Recipes
Describes what recipes are in the data model and the current manufacturing problem.

Modes
Describes what *modes* are in the data model and the current manufacturing problem.

Material production
Describes the material production object the data model and the current manufacturing problem.

Production orders
Describes what *production orders* are in the data model and the current manufacturing problem.

Production to demand arcs
Describes what *pegging arcs*, in particular *production to demand arcs*, represent in the data model and the current manufacturing problem.

Weighted objectives
Describes what *weighted objectives* are in the data model and the current manufacturing problem.

Overview of the process
Describes how the many data model objects are used together to represent the problem data.

Table of data model objects
A table that relates the data schema and C++ elements.

Model the process

The purpose of this section is to introduce some of the building blocks of Plant PowerOps that are necessary to model a problem, such as *Recipes*, *Demands*, *Production orders*, *Modes*, and *Production to demand arcs*.

With these objects of PPO, you model the *process* rather than trying to schedule explicit, individual activities on resources. You define the *recipe* of how a product or material is made. You create the customer demands, due dates, nondelivery costs and revenue that drive the creation of *production orders* that follow your recipes to produce material. The material itself could be a finished product or an intermediate material; most factories would use many recipes to handle the various intermediates and finished products.

Materials

Materials in PPO can specify raw materials, intermediate products consumed in the manufacturing process, and finished products.

In this example, the dried cradles, ready for delivery to a customer, are a material.

Demands

A demand object, in PPO, represents a forecast or a customer request for a certain quantity of *finished* products. A time window for completion can be specified, with an optional preferred due date. The due date is used to compute the earliness and tardiness costs previously discussed. If no time window or due date is specified, by default the demand must simply be satisfied by the time horizon of the problem.

Demands normally have an incentive associated with them, too. That is, if you fail to meet a demand, there might be a nondelivery cost. Conversely, if you meet a demand, you might get revenue. If one demand has a higher revenue than another, then (all else being equal) PPO will make a greater effort to satisfy the higher-revenue demand optimally.

In this example, there are ten demands, corresponding to the three pine and seven teak cradles. There are nondelivery, earliness, and tardiness costs associated with each cradle. It is these demands and their costs (or revenues) that drive the process for PPO to ultimately create the production orders that create the specific explicit activities that manufacture products.

There is typically no need to define the specific demands for raw materials or intermediate products that are consumed during the manufacturing process.

Activities

Activities (performed on resources) are what make the materials and finished products. Activities ultimately satisfy demand.

Although it is possible for you to model each and every activity, a better approach is to model activity types (or *prototypes*), and make PPO do the work of generating and scheduling the explicit activities. PPO generates the explicit activities through recipes and production orders. The recipe contains the activity prototypes, and the production order uses these prototypes as a template to generate the actual, explicit activities that are scheduled to meet demand.

There are two activity prototypes in this example, one for pine drying and one for teak drying. Since we have ten demands in total for dried cradles, PPO will create ten explicit drying activities in the schedule, all based on one of the two activity prototypes. Note that there is no due date associated with the actual activities in this example, but rather the due dates are attached to the *demands* for materials; in this case, dried cradles.

Activities are always performed in modes, which links the activity to the use of a certain resource in a certain method of operation. You can also specify an activity to have a setup requirement.

Resources

The resource for this problem is the dryer, but you can also model resources such as workers, vehicles, and supplies. By default, a resource is available for use throughout the time horizon of the problem, but that can be modified for cleaning, maintenance, or other reasons.

The capacity of the dryer resource is 1 (one), because the activity (drying) consumes 100% of the resource during the activity duration.

Recipes

Recipes are a set of activities that model a production process. They define both the materials used or consumed by the process (raw materials or intermediates), and the materials produced by the process (intermediates or finished products). However the essential point is that rather than containing explicit activities, recipes model activity prototypes and constraints. These are, in turn, used as a template or a mold to generate the explicit activities that produce the output material.

In a larger model of the cradle factory, there would be several recipes. The first recipe might consume raw wood to produce the cut and lathed wood pieces. Another recipe would assemble the cradles. A varnishing recipe would consume unvarnished cradles and the varnishing liquid to produce varnished cradles. The drying recipe would consume varnished cradles and produce dried cradles.

In this simple example, we create only two recipes. One recipe defines an activity prototype for drying pine cradles and one defines an activity prototype for drying teak cradles.

Modes

Activities are always performed in *modes*. A mode can be thought of as the method used to perform an activity; which resource is used, what is the resource capacity required, what is the processing time, and so forth. For example, perhaps an activity can be processed on three different resources; this means there are three (at least) different possible modes for this activity. Further, perhaps one of those resources can perform the activity at two different speeds, "normal" or "rush." In that case, there would be two modes to process the activity on that resource, and four modes overall.

For this simple problem only two modes are really needed; the dryer is either used for 70 minutes to dry pine, or 90 minutes to dry teak.

Material production

This object is very important as it represents the consumption or production of all material. It specifies which activities and modes produce and consume materials, in what quantity, and to/from which storage units (if any). Essentially this object links materials with the activities that produce them. This object along with the recipe object help define recipe (as used in the broad sense of meaning production process) in PPO.

Production orders

A *production order* implements the recipe to ultimately meet demand.

Production orders have a batch size, which is a multiplication factor on a recipe. The batch size is used to compute the quantities of materials produced or consumed by the production order. The batch size is also used to adjust the processing times and costs of activities (if variable processing times and costs have been specified on the modes of the activity prototypes of the recipe).

A production order can be used to produce intermediate materials or finished materials (that is, finished manufactured products). A production order that makes finished goods may partially or wholly satisfy a single demand, or may satisfy several demands. A production order producing only intermediates does not satisfy a PPO demand object.

In this example the batch size is 1.0 so there are ten separate production orders, corresponding to the ten cradles (pine and teak combined) that need to be dried.

Production to demand arcs

An arc (or pegging arc) is a PPO object used to ensure proper linkage between two modeling components; an arc is used to directly ensure proper material flow. There are arcs that link procurements to productions, productions to other productions, and productions to demands.

This example uses an arc between a production order and a demand in order to specify the quantity of material from production to the demand.

There are ten arcs in this example, linking the ten production orders for dried cradles to the ten dried cradle demands.

Note that in this small example, you will be creating the arcs yourself. In a larger example, the production order and arc tables are part of the solution provided from the planning (indirectly) and batching (directly) modules. A flow of material between a production order and a customer demand (a production to demand arc) is modeled as a due date on the producing activity of the corresponding production orders. In this example, you will model the arcs manually.

Weighted objectives

When an Plant PowerOps plan is generated, certain objectives can be calculated or measured to judge the appropriateness or quality of the plan. Each of these predefined objectives can be assigned a *weight* to indicate (multiply) its importance in this problem relative to other criteria. Weighted objectives are a type of Key Performance Indicator (KPI); industries have standard sets of KPIs that they use to evaluate plans. Many KPIs are available with PPO.

In this problem, we will define a *total* earliness cost and a *total* tardiness cost. Note that these costs apply to the entire solution; that is, the earliness and tardiness for all activities in the plan is computed. Contrast this with the fixed and variable costs assigned to individual activities. There will also be a total nondelivery cost that sums the nondelivery costs of all demands that are not met.

Overview of the process

From reading this section, you understand that with PPO you can model a scheduling problem at a high level, using recipes, demands, resources, materials, and costs. There is no need for you to attempt to schedule the explicit activities; instead, you use a recipe to model a process, a demand to drive it, and a production order to implement it.

With PPO, you model the demands of customers or forecasts for certain quantities of finished materials and products. These demands or forecasts have nondelivery costs and revenues that drive the process that ultimately creates production orders, that in turn are used to create the explicit manufacturing activities by cloning the activity prototypes of a recipe. The compatibility and precedence constraints between prototype activities in the recipe are also cloned to the explicit activities. The production order uses a batch size to adjust the resources and processing time used by activities, so that materials are consumed and/or produced at the right level.

Table of data model objects

The following table helps organize the modeling objects and information. Each horizontal line refers to a PPO object or concept we have already discussed; and not by coincidence, each line also refers to the name of a PPO data schema table and PPO C++ class.

Table of PPO Data Objects

Object	Members or Function	Data Schema Table	PPO C++ Class
Model	date origin, time unit, and end max for the model; sets environment, contains model objects	`PPO_MODEL`	`IloMSModel`
Resource	resource name and capacity	`PPO_RESOURCE`	`IloMSResource`
Material	defines raw materials, intermediates, finished products	`PPO_MATERIAL`	`IloMSMaterial`
Recipe	name of recipe used for production	`PPO_RECIPE`	`IloMSRecipe`
Activity	name and recipe association	`PPO_ACTIVITY_PROTO`	`IloMSAbstractActivity`
Mode	required resource capacity and processing time per mode	`PPO_MODE_PROTO`	`IloMSMode`
Material Production	links activities to the materials they produce and consume	`PPO_MATERIAL_PRODUCTION_PROTO`	`IloMSMaterialProduction`
Demand	lists all demands for finished product	`PPO_DEMAND`	`IloMSDemand`
Due Date	due dates and associated cost penalties	`PPO_DUE_DATE`	`IloMSDueDate`
Production Order	recipe used and batch size	`PPO_PRODUCTION_ORDER`	`IloMSProductionOrder`
Production to Demand	links a production	`PPO_PROD_TO_DEMAND_ARC`	`IloMSAbstractMaterialFlowArc`

Object	Members or Function	Data Schema Table	PPO C++ Class
Arc	order to a demand with quantity		
Weighted Objectives	criteria used to measure plan quality	`PPO_CRITERION_WEIGHT`	`IloMSModel::setWeight`

Each object you create with a data table is instantiated into a C++ class member by the PPO engines. However, you do not need to know any C++ in order to get the full benefits of PPO; you have full modeling capabilities with data tables or by using the GUI. Since csv and mdb files are an easily understood method of modeling data, they are ideal for learning new concepts, such as the data model. You can transfer that conceptual knowledge to C++ or Java later, if you wish. Note that the table is a summary only, useful for this cradle factory example. The schema tables and API classes of PPO include a great deal more function than shown here. Full descriptions are available in other publications: the *Plant PowerOps Data Schema* and the *Plant PowerOps C++ Reference Manual*. These objects also have counterparts in Java™ , and information can be found for those classes in the *Plant PowerOps Java Reference Manual*.

A quick tutorial on csv and mdb usage in PPO

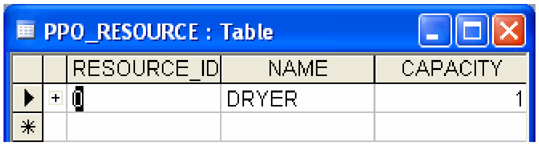
If you are already comfortable with modeling data using csv or mdb files per your choice, you can skip ahead to *Model the problem*.

Otherwise, you might find the following information useful.

Here's the PPO_RESOURCE table shown in csv format:

```
PPO_RESOURCE|NAMES,    RESOURCE_ID,    NAME,     CAPACITY
PPO_RESOURCE|KEYS,     1,              0,        0
PPO_RESOURCE|TYPES,    id,             string,   int
PPO_RESOURCE,          0,              DRYER,    1
```

And as an mdb table:


PPO_RESOURCE : Table

RESOURCE_ID	NAME	CAPACITY
0	DRYER	1

PPO_RESOURCE identifies the name of the table to PPO, and the columns RESOURCE_ID, NAME, and CAPACITY define characteristics of the resource. There is only one resource (DRYER) declared in this example. There are typically many data columns or fields for each table, and you are not likely to use all of the available ones.

Common to both csv and mdb

- ♦ It doesn't matter in which order you define the fields (columns) for a table; you could for example define CAPACITY before NAME in the PPO_RESOURCE table. However the columns must maintain integrity; that is, the column NAME must be defined as a string not an int, and that column must contain the data for NAME, which in this example is DRYER.
- ♦ The data type id is a string, so be careful that you do not accidently add blank white space to that field.
- ♦ In the context of data types used for PPO, you can interchangeably use int or integer, and float or double. For boolean values use 1 and 0 for true and false.
- ♦ You can find more information about KEYS and TYPES in the data schema.
- ♦ You do not have to define all the available fields for a table; you must only declare the fields that are identified as mandatory in the data schema. Be aware however that there are default values for most fields, whether the field is declared in the table or not. Also if you do define a column, you must provide a value for that column in all data rows in the table.

Keys and data types in mdb files

With mdb files, you don't define the keys and data type in the actual data model tables; these are defined in the **Design** view of each table of Microsoft® Access® . See Access help for more information on defining data types and keys.

More about using csv files

- The first three lines in a csv table are used to define the data for PPO. All tables (except for the `PPO_MODEL` table) must start with lines that define the `NAMES`, `KEYS`, and `TYPES` for each field. The `PPO_MODEL` table requires only `NAMES` and `TYPES`.
- The `NAMES` line always starts with the table name (for example `PPO_RESOURCE`) followed immediately by a vertical separator ("`|`") and the keyword `NAMES`. Then you list the fields you are defining for this table, separated by commas.
- The `KEYS` line always starts with the table name followed immediately by a vertical separator ("`|`") and the keyword `KEYS`. Then you indicate whether the field is a key for the table ("`1`") or not ("`0`").
- The `TYPES` line always starts with the table name followed immediately by a vertical separator ("`|`") and the keyword `TYPES`. Then indicate what data type the field is, such as `boolean` or `float`.
- The line following the `TYPES` line is always used to start entering data.
- In the following examples, the extra spaces in the tables are not necessary; we used extra spaces here just to create orderly, legible columns.

Model the problem

Now you have a problem description and know something about the PPO data model. Let's begin to model the problem by gathering the necessary data.

Data checklist

Use this checklist to gather data for modeling most PPO problems.

- ♦ What is the date origin, time horizon, and time unit for this problem?
- ♦ What are the resources and their capacities for this problem?
- ♦ What are the types of activities in this problem?
- ♦ What are the recipes by which these activities are manufactured?
- ♦ What are the materials produced by these recipes?
- ♦ What are the production orders that drive the activities?
- ♦ What are the customer demands that drive the production orders?
- ♦ What are the due dates for these demands, and any associated cost factors?
- ♦ What are the weighted objectives for this problem?

We can answer these questions and, with a little help from the *Data Schema*, can now model the problem. The *Data Schema* is used to ensure that the model has data integrity. Correct use of the schema ensures that PPO will interpret the problem data in the manner you intended.

The files that we build in the following steps contains all the necessary data tables to completely model the problem. We'll demonstrate simultaneously building the model in csv and mdb format. You can see the complete examples in <*PPOInstallDirectory*>`\doc\chm\IGLesson1_Basic.csv` and <*PPOInstallDirectory*>`\examples\data\course\CourseLesson1_Basic.mdb` in your development environment. In some example files, you might find additional data that is not discussed here; generally this is default data and you can refer to the *Data Schema* for more information.

Step 1 Declare the schema

All Plant PowerOps csv files start with two lines specifying the format and schema versions. The format is `1` and the schema is at version `2`.

```
ILOG_CSV_FORMAT,   1
ILOG_DATA_SCHEMA,  PPO,   4.0
```

The PPO data schema is a public schema, and every csv file you build starts with these two lines. (There is no corresponding table in mdb format.)

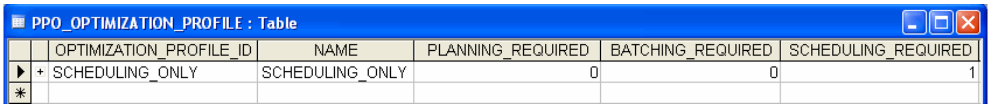
Step 2 Define the optimization profile

This step is not strictly necessary for this example. However, in more complicated problems it's normal to use the PPO_OPTIMIZATION_PROFILE table to control or tune the optimization engines of PPO. You can learn more at *Optimization profile*. For now, just know that this example is much smaller than a typical PPO problem and is designed to be scheduled within a two-day period. So we don't need all of the power of PPO; we will only use the scheduling module, not the planning and batching modules. The following csv table defines a profile called SCHEDULING_ONLY.

```
PPO_OPT_PRO|NAMES, OPTIMIZATION_PROFILE_ID, NAME,              PLANNING
 BATCHING      SCHEDULING
                                                                _REQUIRED,
 _REQUIRED,     _REQUIRED
PPO_OPT_PRO|KEYS,  1,                        0,                 0,
 0,             0
PPO_OPT_PRO|TYPES, id,                       string,            boolean,
 boolean,       boolean
PPO_OPT_PRO,       SCHEDULING_ONLY,          SCHEDULING_ONLY,   0,
 0,             1
```

Note that for presentation purposes in this manual, we have abbreviated the table name to PPO_OPT_PRO and stacked the NAMES row into two lines. Do not do either of these actions in your actual csv file!

Here is the corresponding table in Microsoft® Office Access® .mdb format:



PPO_OPTIMIZATION_PROFILE : Table

OPTIMIZATION_PROFILE_ID	NAME	PLANNING_REQUIRED	BATCHING_REQUIRED	SCHEDULING_REQUIRED
SCHEDULING_ONLY	SCHEDULING_ONLY	0	0	1

Step 3 Create the environment

With this step, we finally start to build the actual problem data model. The first thing to do is define the problem environment with the PPO_MODEL table; fields include the problem NAME, INT_DATE_ORIGIN, TIME_UNIT, END_MAX (time horizon), and the START_MIN. The start minimum field defines the common earliest start time for activities requiring plant resources that need to be scheduled (does not include activities that are known to have already started). We set START_MIN to zero, thus setting it equal to the date origin.

```
PPO_MODEL|NAMES, NAME,         INT_DATE_ORIGIN, TIME_UNIT, START_MIN, END_MAX,
  CURRENT

   _OPTIMIZATION

   _PROFILE
PPO_MODEL|TYPES, string,        int,              int,        int,        int,
   id
```

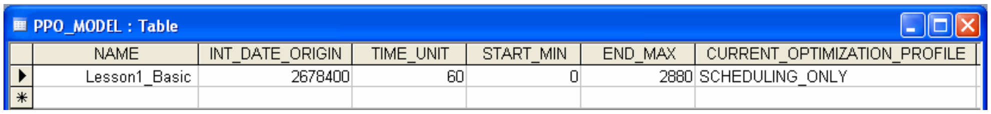
```
PPO_MODEL, CourseLesson1Basic, 2678400,        60,         0,          2880,
   SCHEDULING_ONLY
```

The `INT_DATE_ORIGIN` is used to express the date origin in seconds since January 1, 2001. The value here is `2678400` seconds (31*24*60*60) or February 1, 2001, 00:00 GMT. Alternatively, one can use a field called `DATE_ORIGIN` to express the starting point in "`YYYY-MM-DD HH:MM:SS`" format. Note that although the `INT_DATE_ORIGIN` is expressed in seconds, the `START_MIN` and `END_MAX` fields are expressed in the declared time unit (minutes).

Also, note that a `CURRENT_OPTIMIZATION_PROFILE` for this model is declared: `SCHEDULING_ONLY`, the profile we created in the step immediately prior.

Although not part of this example, this table is also used to define the time zone. The `INT_DATE_ORIGIN` field is given in the defined time zone; whereas the `DATE_ORIGIN` is provided in UTC. For more information see *The model environment* and *Time zone settings*.

Here is the corresponding table in .mdb format:



PPO_MODEL : Table

NAME	INT_DATE_ORIGIN	TIME_UNIT	START_MIN	END_MAX	CURRENT_OPTIMIZATION_PROFILE
Lesson1_Basic	2678400	60	0	2880	SCHEDULING_ONLY

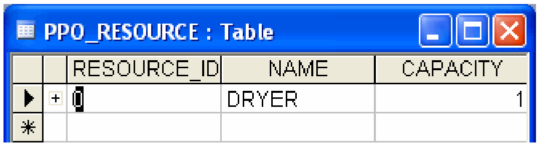
Step 4 Define the resource

The following lines create the `PPO_RESOURCE` table, which defines the `RESOURCE_ID`, `NAME`, and `CAPACITY` of the `DRYER` resource. Because the problem definition involves a single machine, only a single resource is required for this example.

```
PPO_RESOURCE|NAMES,    RESOURCE_ID,    NAME,     CAPACITY
PPO_RESOURCE|KEYS,     1,              0,        0
PPO_RESOURCE|TYPES,    id,             string,   int
PPO_RESOURCE,          0,              DRYER,    1
```

The `CAPACITY` for this machine, because each activity requires 100% of the resource, is set to 1 (an indivisible integer). The default value for the capacity field is one, so in fact this field is not necessary for this example; we model it here just to highlight its use. Another useful field in this table is `DISPLAY_RANK` which orders how resources appear as listed in the GUI; this field is not necessary since there is only one resource in this problem.

Here is the mdb resource table:



PPO_RESOURCE : Table

RESOURCE_ID	NAME	CAPACITY
0	DRYER	1

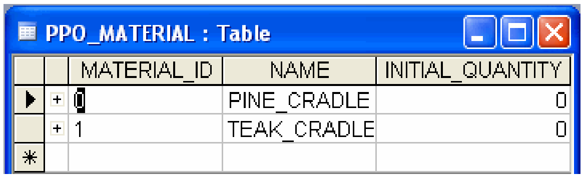
Step 5 Define the materials

The next object to define is material. Remember that materials in Plant PowerOps can be raw materials, intermediate products, or finished products. In this example we model only two finished products—PINE_CRADLES and TEAK_CRADLES. The following lines create the PPO_MATERIAL table:

```
PPO_MATERIAL|NAMES,    MATERIAL_ID, NAME
PPO_MATERIAL|KEYS,     1,           0
PPO_MATERIAL|TYPES,    id,          string
PPO_MATERIAL,          0,           PINE_CRADLE
PPO_MATERIAL,          1,           TEAK_CRADLE
```

Note that we define the materials according to ascending order of their MATERIAL_ID to make it easier to read the data.

In the database file, we also declare an initial quantity of zero; this is not strictly necessary since that is the default for this field.


PPO_MATERIAL : Table

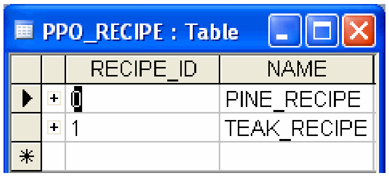
MATERIAL_ID	NAME	INITIAL_QUANTITY
0	PINE_CRADLE	0
1	TEAK_CRADLE	0

Step 6 Define the recipe

Recipes are defined as a set of activities, the combination of processes and materials required for production. In this example we consider only the production of dried cradles. The following lines create the PPO_RECIPE table:

```
PPO_RECIPE|NAMES,      RECIPE_ID,   NAME
PPO_RECIPE|KEYS,       1,           0
PPO_RECIPE|TYPES,      id,          string
PPO_RECIPE,            0,           PINE_RECIPE
PPO_RECIPE,            1,           TEAK_RECIPE
```

There are many other fields available in the PPO_RECIPE table, but we either do not need to use them or the defaults are fine.


PPO_RECIPE : Table

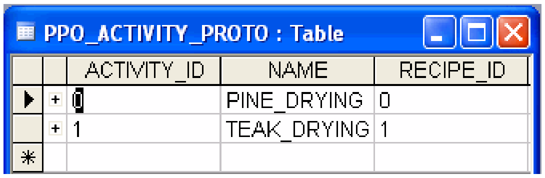
RECIPE_ID	NAME
0	PINE_RECIPE
1	TEAK_RECIPE

Step 7 Define the activity prototypes

Use the PPO_ACTIVITY_PROTO table to define two *types* of activities, or activity prototypes, corresponding to the two activity types in our example: PINE_DRYING and TEAK_DRYING. Note that if we were defining all the actual activities of this example—drying three pine and seven teak cradles—we would have ten activity instances to define just in this simple problem.

You also declare the RECIPE_ID to which the activity prototype belongs.

```
PPO_ACTIVITY_PROTO|NAMES,   ACTIVITY_ID, NAME,        RECIPE_ID
PPO_ACTIVITY_PROTO|KEYS,    1,           0,           0
PPO_ACTIVITY_PROTO|TYPES,   id,          string,      id
PPO_ACTIVITY_PROTO,         0,           PINE_DRYING, 0
PPO_ACTIVITY_PROTO,         1,           TEAK_DRYING, 1
```


PPO_ACTIVITY_PROTO : Table

ACTIVITY_ID	NAME	RECIPE_ID
0	PINE_DRYING	0
1	TEAK_DRYING	1

Step 8 Define mode prototypes for the activity prototypes

In the PPO_MODE_PROTO table we use two primary key fields: ACTIVITY_ID and MODE_NUMBER. This table associates an ACTIVITY_ID and MODE_NUMBER pair (identified by the mode NAME) with a resource and a processing time. In this example, there is only one mode for each activity, so each activity needs only one MODE_NUMBER (0).

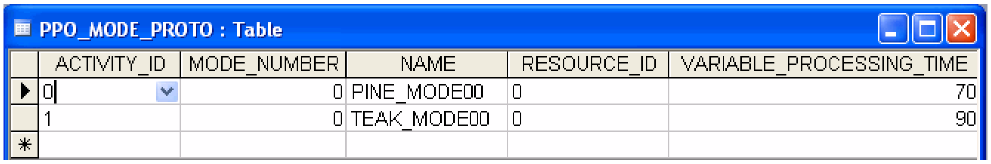
Also, remember that since we defined activity prototypes in the PPO_ACTIVITY_PROTO table rather than individual activities, the ACTIVITY_ID in this table refers to an activity type, and so we are now defining the mode for an activity type.

```
PPO_MODE_PROTO|NAMES, ACTIVITY_ID, MODE_NUMBER, NAME,        RESOURCE_ID,
VARIABLE_PROCESSING_TIME
PPO_MODE_PROTO|KEYS,  1,           1,           0,           0,           0
PPO_MODE_PROTO|TYPES, id,         int,         string,       id,          float
PPO_MODE_PROTO,       0,           0,           PINE_MODE00, 0,           70.
0
PPO_MODE_PROTO,       1,           0,           TEAK_MODE00, 0,           90.
0
```

The VARIABLE_PROCESSING_TIME field is used to define the different processing times for pine cradles (70 minutes) and teak cradles (90 minutes).

The dryer resource can be used by only one activity at a time—each activity requires 100% of the capacity of the resource. The required capacity for both modes is one; since this is the default for the REQUIRED_CAPACITY field, we don’t have to include it in the table.

As previously mentioned, in Microsoft Access the keys for a table are defined in the Design view. Here is the mdb table shown in the table view.


PPO_MODE_PROTO : Table

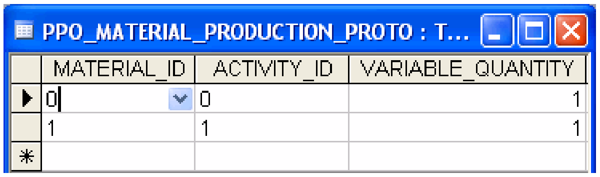
ACTIVITY_ID	MODE_NUMBER	NAME	RESOURCE_ID	VARIABLE_PROCESSING_TIME
0	0	PINE_MODE00	0	70
1	0	TEAK_MODE00	0	90

Step 9 Define the material production

The following lines create the PPO_MATERIAL_PRODUCTION_PROTO table, which pairs an activity with material production or consumption. The quantity produced (or consumed, if the value is negative) can be variable (quantity produced for the execution of one unit of recipe) or fixed (quantity produced for the execution of one batch of recipe). In this example, we are linking an activity prototype with material production, with a variable quantity of one.

```
PPO_MATERIAL_PRODUCTION_PROTO|NAMES, MATERIAL_ID, ACTIVITY_ID, VARIABLE_QUANTITY
PPO_MATERIAL_PRODUCTION_PROTO|KEYS,  1,           1,           0
PPO_MATERIAL_PRODUCTION_PROTO|TYPES, id,          id,          float
PPO_MATERIAL_PRODUCTION_PROTO,       0,           0,           1.0
PPO_MATERIAL_PRODUCTION_PROTO,       1,           1,           1.0
```

We have modes defined in this problem, yet there is no declaration regarding material production of the modes in this table. The reason is that the default value for the MODE_NUMBER field is a special value: -1 (minus one). This value declares that all modes of the activity produce or consume the material in the same manner. When either activity is performed, it always produces one unit as defined in the table.


PPO_MATERIAL_PRODUCTION_PROTO : T...

MATERIAL_ID	ACTIVITY_ID	VARIABLE_QUANTITY
0	0	1
1	1	1

This table is also used to identify any storage activities and units for the produced materials, but these are not modeled in this problem. You can also specify if material is produced continuously or in batch (default).

Step 10 Define the demand

The PPO_DEMAND table represents the request for a specified QUANTITY of *finished* material (MATERIAL_ID) deliverable in a time window, with an optional preferred due date. Do not use this table to specify demand for intermediate products that are consumed by the manufacturing process.

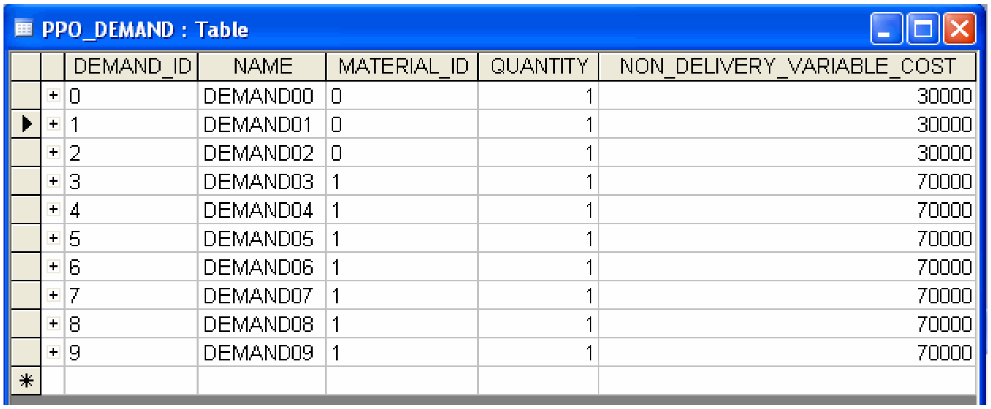
In this example there are three demands for pine cradles (MATERIAL_ID 0, set in the PPO_MATERIAL table) and seven demands for teak cradles (MATERIAL_ID 1).

```
PPO_DEMAND|NAMES,      DEMAND_ID,    NAME,         MATERIAL_ID, QUANTITY,
NON_DELIVERY_VARIABLE_COST
```

```
PPO_DEMAND|KEYS,       1,          0,          0,          0,        0
PPO_DEMAND|TYPES,      id,         string,     id,         float,    float
PPO_DEMAND,            0,          DEMAND00,   0,          1.0,      30000
PPO_DEMAND,            1,          DEMAND01,   0,          1.0,      30000
PPO_DEMAND,            2,          DEMAND02,   0,          1.0,      30000
PPO_DEMAND,            3,          DEMAND03,   1,          1.0,      70000
PPO_DEMAND,            4,          DEMAND04,   1,          1.0,      70000
PPO_DEMAND,            5,          DEMAND05,   1,          1.0,      70000
PPO_DEMAND,            6,          DEMAND06,   1,          1.0,      70000
PPO_DEMAND,            7,          DEMAND07,   1,          1.0,      70000
PPO_DEMAND,            8,          DEMAND08,   1,          1.0,      70000
PPO_DEMAND,            9,          DEMAND09,   1,          1.0,      70000
```

As described earlier, the demands have a cost for failing to deliver: the NON_DELIVERY_VARIABLE_COST. Note that the nondelivery costs for material one (the teak cradles) are higher than for pine. This provides a higher incentive for PPO to meet these demands.

A delivery time window that defines the earliest and latest possible delivery times can be specified with the fields DELIVERY_START_MIN and DELIVERY_END_MAX. These fields default to -INF and +INF. We don't declare the time window in this example, but we will have a due date. The due date for each demand is defined in the next step.



DEMAND_ID	NAME	MATERIAL_ID	QUANTITY	NON_DELIVERY_VARIABLE_COST
0	DEMAND00	0	1	30000
1	DEMAND01	0	1	30000
2	DEMAND02	0	1	30000
3	DEMAND03	1	1	70000
4	DEMAND04	1	1	70000
5	DEMAND05	1	1	70000
6	DEMAND06	1	1	70000
7	DEMAND07	1	1	70000
8	DEMAND08	1	1	70000
9	DEMAND09	1	1	70000

Step 11 Define due dates for the demands

This example implements a "customer demand driven" model for this problem, so the PPO_DUE_DATE table is used to associate a due date (DUE_TIME) with each of the demands defined in the PPO_DEMAND table (through DEMAND_ID).

This table is also used to assign earliness and tardiness fixed costs, and earliness and tardiness variable costs. In this example there are no fixed costs.

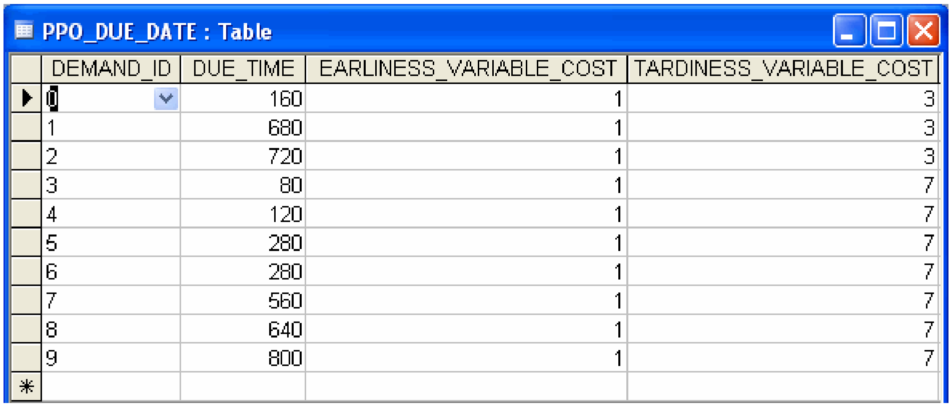
```
PPO_DUE_DATE|NAMES,  DEMAND_ID,    DUE_TIME,    EARLINESS_VARIABLE_COST,
TARDINESS_VARIABLE_COST
PPO_DUE_DATE|KEYS,   1,            0,           0,                          0
PPO_DUE_DATE|TYPES,  id,          int,         float,                      float
```

```
PPO_DUE_DATE,          0,           160,        1.0,                        3.0
PPO_DUE_DATE,          1,           680,        1.0,                        3.0
PPO_DUE_DATE,          2,           720,        1.0,                        3.0
PPO_DUE_DATE,          3,            80,        1.0,                        7.0
PPO_DUE_DATE,          4,           120,        1.0,                        7.0
PPO_DUE_DATE,          5,           280,        1.0,                        7.0
PPO_DUE_DATE,          6,           280,        1.0,                        7.0
PPO_DUE_DATE,          7,           560,        1.0,                        7.0
PPO_DUE_DATE,          8,           640,        1.0,                        7.0
PPO_DUE_DATE,          9,           800,        1.0,                        7.0
```

You can see that the due time is expressed in terms of the time units after the date origin of the problem.

Attaching a due date to a demand, rather than to specific activities, is a key concept of the modeling methodology to use activity prototypes and thereby allowing PPO to do the hard work of creating all the specific necessary scheduled activities.

The due date table in mdb format:


PPO_DUE_DATE : Table

DEMAND_ID	DUE_TIME	EARLINESS_VARIABLE_COST	TARDINESS_VARIABLE_COST
0	160	1	3
1	680	1	3
2	720	1	3
3	80	1	7
4	120	1	7
5	280	1	7
6	280	1	7
7	560	1	7
8	640	1	7
9	800	1	7

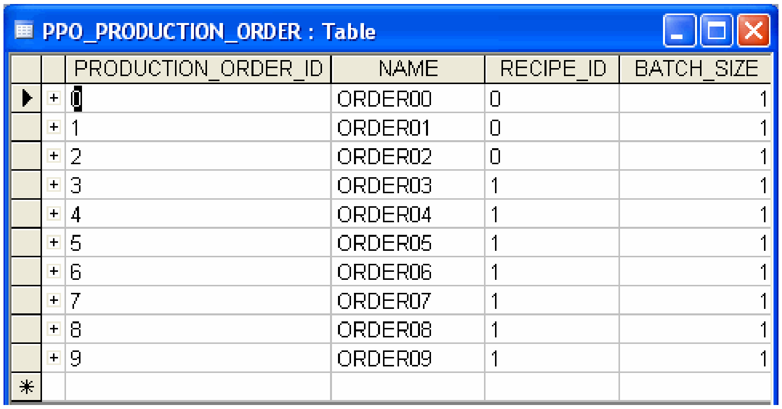
Step 12 Define the production orders

The `PPO_PRODUCTION_ORDER` table represents a production order implementing the recipe of a process. Notice that there are three production orders for the recipe producing pine cradles (`RECIPE_ID 0`) and seven production orders for the recipe producing teak cradles (`RECIPE_ID 1`). The following lines create the `PPO_PRODUCTION_ORDER` table.

```
PPO_PRODUCTION_ORDER|NAMES, PRODUCTION_ORDER_ID,  NAME,     RECIPE_ID,
BATCH_SIZE
PPO_PRODUCTION_ORDER|KEYS,  1,                    0,        0,          0
PPO_PRODUCTION_ORDER|TYPES, id,                   string,   id,         float
PPO_PRODUCTION_ORDER,       0,                    ORDER00,  0,          1.0
PPO_PRODUCTION_ORDER,       1,                    ORDER01,  0,          1.0
PPO_PRODUCTION_ORDER,       2,                    ORDER02,  0,          1.0
PPO_PRODUCTION_ORDER,       3,                    ORDER03,  1,          1.0
```

```
PPO_PRODUCTION_ORDER,        4,                    ORDER04,  1,          1.0
PPO_PRODUCTION_ORDER,        5,                    ORDER05,  1,          1.0
PPO_PRODUCTION_ORDER,        6,                    ORDER06,  1,          1.0
PPO_PRODUCTION_ORDER,        7,                    ORDER07,  1,          1.0
PPO_PRODUCTION_ORDER,        8,                    ORDER08,  1,          1.0
PPO_PRODUCTION_ORDER,        9,                    ORDER09,  1,          1.0
```

The `BATCH_SIZE` column is optional in this example, as the default is `1.0`.



PPO_PRODUCTION_ORDER : Table

PRODUCTION_ORDER_ID	NAME	RECIPE_ID	BATCH_SIZE
0	ORDER00	0	1
1	ORDER01	0	1
2	ORDER02	0	1
3	ORDER03	1	1
4	ORDER04	1	1
5	ORDER05	1	1
6	ORDER06	1	1
7	ORDER07	1	1
8	ORDER08	1	1
9	ORDER09	1	1

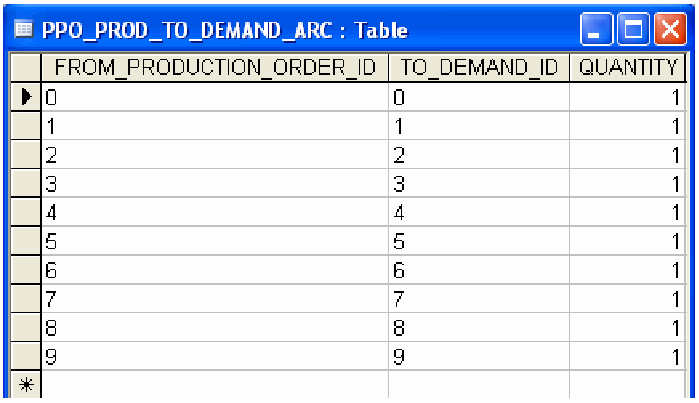
Step 13 Link the production order to the demands

A link (or arc) for the flow of material from a production order to a demand is made with the `PPO_PROD_TO_DEMAND_ARC` table, which uses both the production order and the demand as primary keys.

A `QUANTITY` is defined in this table for this production order. The value of `QUANTITY` in this table does not have to be the same as the value of `QUANTITY` in the `PPO_DEMAND` table; for example, there might be multiple production orders required to satisfy a single demand.

```
PPO_PROD_TO_DEMAND_ARC|NAMES, FROM_PRODUCTION_ORDER_ID,  TO_DEMAND_ID,
QUANTITY
PPO_PROD_TO_DEMAND_ARC|KEYS,  1,                          1,              0
PPO_PROD_TO_DEMAND_ARC|TYPES, int,                       int,            float
PPO_PROD_TO_DEMAND_ARC,       0,                          0,              1.0
PPO_PROD_TO_DEMAND_ARC,       1,                          1,              1.0
PPO_PROD_TO_DEMAND_ARC,       2,                          2,              1.0
PPO_PROD_TO_DEMAND_ARC,       3,                          3,              1.0
PPO_PROD_TO_DEMAND_ARC,       4,                          4,              1.0
PPO_PROD_TO_DEMAND_ARC,       5,                          5,              1.0
PPO_PROD_TO_DEMAND_ARC,       6,                          6,              1.0
PPO_PROD_TO_DEMAND_ARC,       7,                          7,              1.0
PPO_PROD_TO_DEMAND_ARC,       8,                          8,              1.0
PPO_PROD_TO_DEMAND_ARC,       9,                          9,              1.0
```

Since this problem is a simple scheduling problem, you create this table and the production order table yourself. In larger problems, the production order and arc tables are typically generated by the planning and batching modules of PPO, and supplied to the scheduling module.



PPO_PROD_TO_DEMAND_ARC : Table

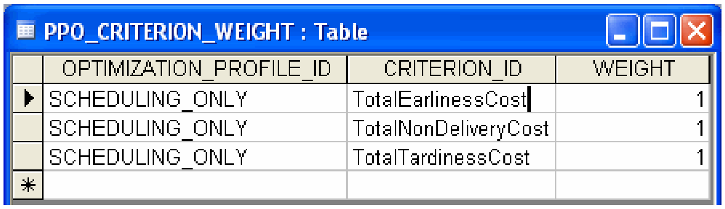
FROM_PRODUCTION_ORDER_ID	TO_DEMAND_ID	QUANTITY
0	0	1
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	1
7	7	1
8	8	1
9	9	1

Step 14 Define the weighted objectives

The `PPO_CRITERION_WEIGHT` table defines the global weights for the three optimization criteria declared for this problem. Optimization criteria can be associated with a specific optimization profile, and we use them in the `SCHEDULING_ONLY` profile here.

```
PPO_CRITERION_WEIGHT|NAMES, OPTIMIZATION_PROFILE_ID, CRITERION_ID,
WEIGHT
PPO_CRITERION_WEIGHT|KEYS,  1,                       1,                 0
PPO_CRITERION_WEIGHT|TYPES, id,                      id,
float
PPO_CRITERION_WEIGHT,       SCHEDULING_ONLY,         TotalNonDeliveryCost, 1.
0
PPO_CRITERION_WEIGHT,       SCHEDULING_ONLY,         TotalEarlinessCost,   1.
0
PPO_CRITERION_WEIGHT,       SCHEDULING_ONLY,         TotalTardinessCost,   1.
0
```

We define criteria that sum the total nondelivery, earliness, and tardiness costs in a generated schedule. Optimization criteria are a specific type of Key Performance Indicator (KPIs); for more information see *Key Performance Indicators*. Here is the mdb table:



PPO_CRITERION_WEIGHT : Table

OPTIMIZATION_PROFILE_ID	CRITERION_ID	WEIGHT
SCHEDULING_ONLY	TotalEarlinessCost	1
SCHEDULING_ONLY	TotalNonDeliveryCost	1
SCHEDULING_ONLY	TotalTardinessCost	1

This is the final step in creating the data files for this example. We now have our complete problem data model. It’s time to solve the problem. Use the sample provided in

<*PPOInstallDirectory*>`/doc/chm/IGLesson1_Basic.csv` in your development environment to solve the model to this problem.

Solving

Describes how to use PPO to solve the problem data previously encoded.

In this section

Overview
Describes the options available for solving the problem.

Solve in the PPO GUI
A review of how to solve for experienced users of PPO.

Overview

You have a choice when it comes to solving the problem model. You can load the csv or mdb model into the PPO GUI, and solve it there. You can use the csv tables as data and solve using the C++ or Java™ API.

In either case, the solution includes the start and end times for the drying activities, and any associated earliness, tardiness, and nondelivery costs. PPO generates a wealth of other data regarding the manufacturing plan, and this is readily available in the GUI.

It’s quite easy to solve the problem model in the PPO GUI. Instructions follow in the next section *Solve in the PPO GUI*.

If you prefer to solve using one of the APIs, see *Using the PPO API for C++ to model and solve* or *Using the PPO API for Java to model and solve*.

Solve in the PPO GUI

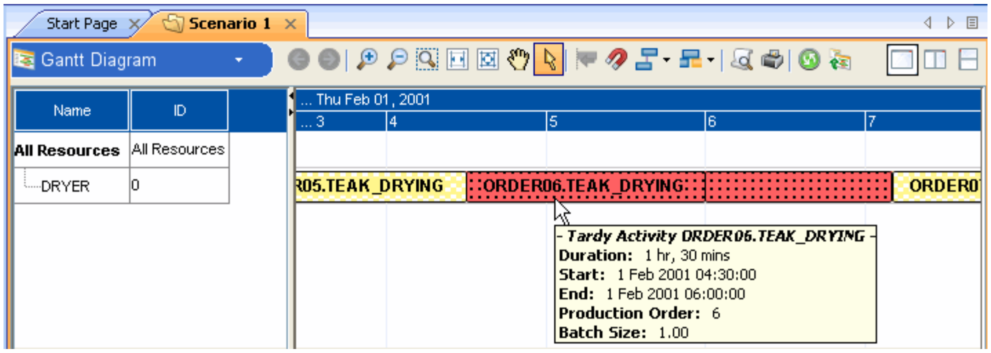
Start the PPO GUI, open the file `IGLesson1_Basic.csv`, and select the **Optimize the scenario** icon on the toolbar. ▶

On the **Optimize the scenario** dialog box, make sure that the `SCHEDULING_ONLY` profile is selected (only the **Detailed Scheduling** module will be checked). Then select **Optimize the scenario**. When scheduling is completed select **Close** on the optimization progress dialog box to view the plan.

Continue with *View and study the plan*.

View and study the plan

The image below shows the **Gantt Diagram** generated by PPO for `IGLesson1_Basic.csv`.



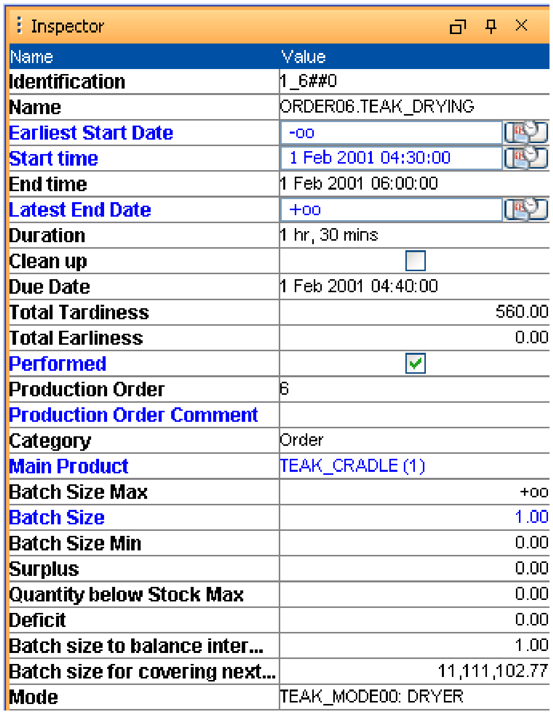
- Tardy Activity ORDER06.TEAK_DRYING -
Duration: 1 hr, 30 mins
Start: 1 Feb 2001 04:30:00
End: 1 Feb 2001 06:00:00
Production Order: 6
Batch Size: 1.00

.

Important: Screen shots shown here may not always exactly match the results you see due to a variety of factors.

The image shows the resource **DRYER** on the left, and all the activities that are scheduled on that resource are in the Gantt chart to the right, according to time slot. The pointer is hovering over an activity, and a tooltip displays information; notably, you can see that this activity is scheduled later than its due date (tardy).

In the menu bar select **Tools > Inspector**, then double-click the activity in the Gantt Diagram. The **Inspector** reveals details about the activity and associated production order. The tardiness cost for this activity is 560.

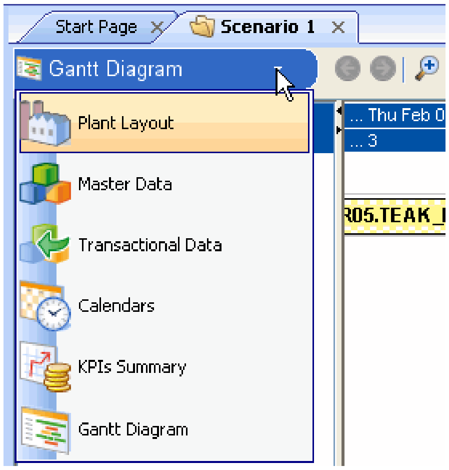


Inspector

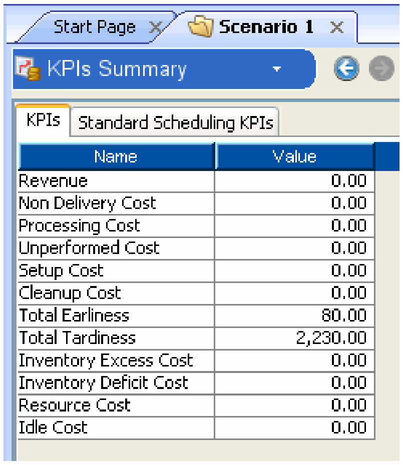
Name	Value
Identification	1_6##0
Name	ORDER06.TEAK_DRYING
Earliest Start Date	-oo
Start time	1 Feb 2001 04:30:00
End time	1 Feb 2001 06:00:00
Latest End Date	+oo
Duration	1 hr, 30 mins
Clean up	☐
Due Date	1 Feb 2001 04:40:00
Total Tardiness	560.00
Total Earliness	0.00
Performed	✓
Production Order	6
Production Order Comment	
Category	Order
Main Product	TEAK_CRADLE (1)
Batch Size Max	+oo
Batch Size	1.00
Batch Size Min	0.00
Surplus	0.00
Quantity below Stock Max	0.00
Deficit	0.00
Batch size to balance inter...	1.00
Batch size for covering next...	11,111,102.77
Mode	TEAK_MODE00: DRYER

The items in blue indicate fields that you can change. So it's possible to change the **Start time**, **Batch Size**, and make a **Comment** about this activity. In fact, you can edit much of the data that you originally input with data tables, either with the Inspector or in the **Master Data** view.

The **Master Data** view is available by clicking **Gantt Diagram**, which displays the view types bar.



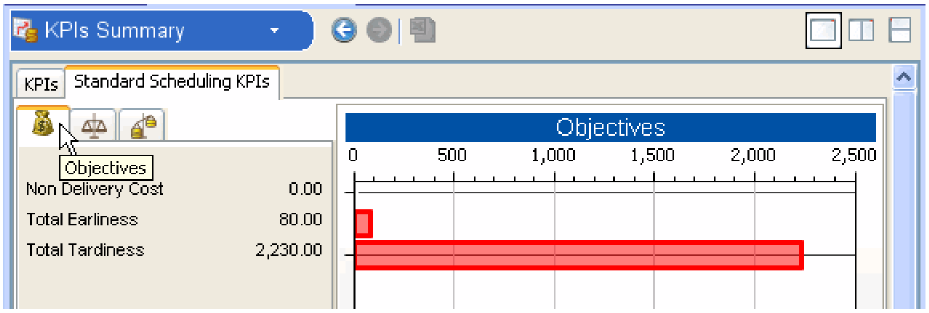
Let's look at another view, the **KPIs Summary** view.



Name	Value
Revenue	0.00
Non Delivery Cost	0.00
Processing Cost	0.00
Unperformed Cost	0.00
Setup Cost	0.00
Cleanup Cost	0.00
Total Earliness	80.00
Total Tardiness	2,230.00
Inventory Excess Cost	0.00
Inventory Deficit Cost	0.00
Resource Cost	0.00
Idle Cost	0.00

Three "weighted criteria" were defined in this example, and here they are with the calculated values for this plan. **Total Earliness** and **Total Tardiness** have some significant costs associated with them, but **Non Delivery Cost** is zero; we can conclude that all demands must have been met, even if early or late. The other criteria displayed here were not defined in our data model, so are zero.

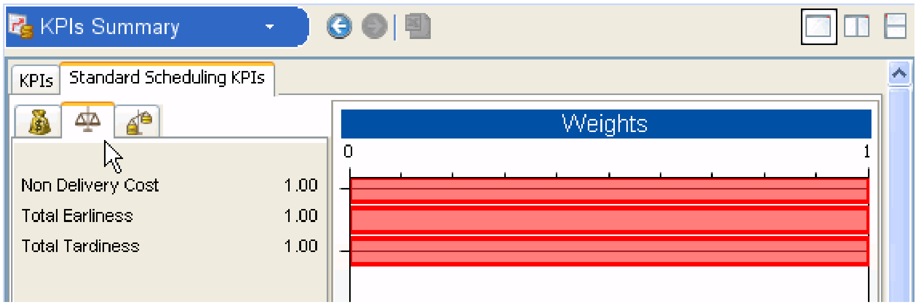
We can take a closer look at these costs by selecting the **Standard Scheduling KPIs** tab. On this page, select the money bag sign, which displays the **Objectives**.



Non Delivery Cost	0.00
Total Earliness	80.00
Total Tardiness	2,230.00

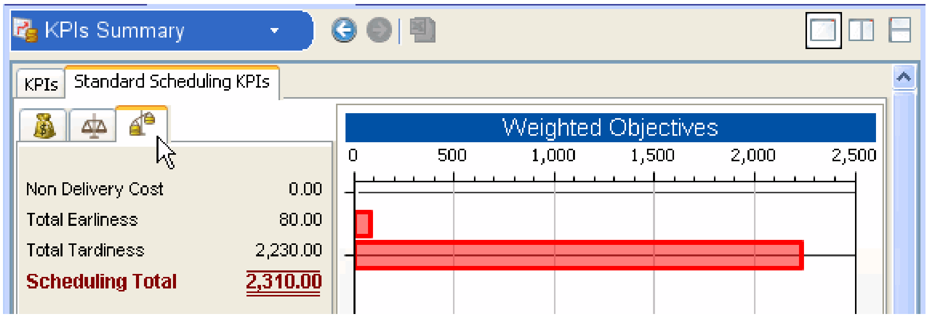
These values are the "raw" cost values from the plan. The tardiness is **2230.00**, and of this, the activity we examined a few steps ago with the Inspector window contributed **560.00**.

Next select the even balance scale to display the **Weights**. These are the values set in the `PPO_CRITERION_WEIGHT` data table; all are equal to **1.00**.



Non Delivery Cost	1.00
Total Earliness	1.00
Total Tardiness	1.00

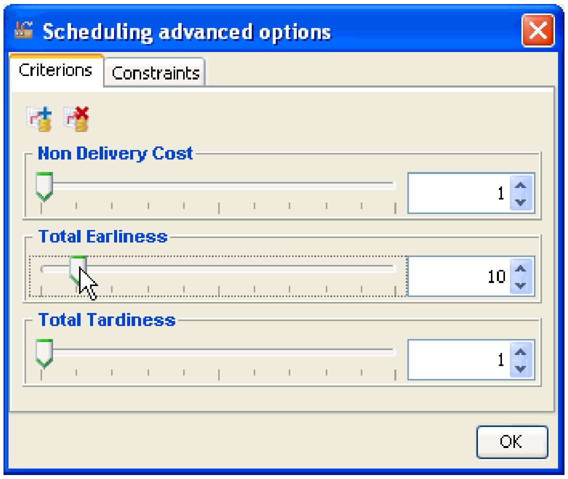
Finally, click the **Weighted Objectives** tab (the last one). You see the results of the values on the first page (raw costs, or objectives) multiplied by the values on the second page (weights).



Non Delivery Cost	0.00
Total Earliness	80.00
Total Tardiness	2,230.00
Scheduling Total	2,310.00

All the weighted objectives that you specified are added up to get the **Scheduling Total**; and a major goal of the PPO scheduling module is to minimize that number. So, the values used in setting criteria are quite important.

You might want to change these criteria values very quickly at times. Changing the value in the data file, then reloading the file and optimizing again is not convenient when you want to quickly run simulations. So PPO provides a quicker way. When you are optimizing and the **Optimize the scenario** dialog box is displayed, select **Advanced options** for the scheduling module. The following window appears.



Non Delivery Cost: 1

Total Earliness: 10

Total Tardiness: 1

Here the user has increased the **Total Earliness** weight to **10**. This will significantly increase the impact of earliness costs in the scheduling module; this action is likely to create a different solution in the next optimization. (When optimizing a plan more than once, note that you have to create a new scenario in the GUI to prevent overwriting the first plan.)

There are a number of additional ways for you to modify the generated plan, or the problem data within the GUI, and regenerate a plan. This makes PPO ideal for doing "what-if" analysis, for example. You can find more information in the documentation set and the example files in the standard delivery.

Review

In this chapter you were introduced to concepts and components of the Plant PowerOps Data Model. You learned how to use:

- The Plant PowerOps environment,
- Activities,
- Resources,
- Materials,
- Modes,
- Recipes,
- Production orders,
- Demands and due dates,
- Weighted objectives, and
- Tables used to link production to demand, and material production to activities and recipes.

You also learned how to create and use optimization profiles.

You used these concepts to:

- Analyze and Describe a manufacturing problem,
- Create a Plant PowerOps model of the problem using csv and mdb tables.
- Solve the problem and view the solution using the PPO GUI.

You’ve been introduced to the basics of the PPO Data Model and have some experience using data tables to model a problem. The next chapter builds upon this knowledge and introduces new problem data for you to model.

Using the PPO API for C++ to model and solve

The first section describes how to use C++ to solve a problem model built with csv database files. The second section describes how to use the C++ API to model and solve a scheduling problem that uses calendars. You may also wish to refer to *Using PPO with Microsoft products*.

In this section

Solving using csv files with C++
Describes how to solve a problem model using C++.

Model and solve using the API for C++
This section describes in detail how to model and solve a manufacturing problem using the PowerOps API for C++.

Solving using csv files with C++

Plant PowerOps csv files can be called from C++ programs as an alternative to solving in the GUI. A sample C++ program `csvsch.cpp` is provided (located at *<PPOInstallDirectory>*`/examples/src/csvsch.cpp`) which you can compile, and the resulting executable used to input csv files to Plant PowerOps.

Once you have compiled the `csvsch.cpp` program, the calling syntax for the resulting executable is as follows:

```
csvsch CourseLesson1_Basic.csv
```

That is, the name of the executable (`csvsch` or `csvsch.exe`), followed by the name of the csv file you are running, with or without its `.csv` extension (for example, `CourseLesson1_Basic`).

The `csvsch` executable can be used to generate plans for the other examples simply by changing the name of the csv file specified as an argument.

Running this program provides output similar to the following, which can be used to assess the generated plan.

```
* Start solving
*
* Time limit     : 10
*
*      no    time         tot_earliness        tot_tardiness           total
*
*  !    1    0.0                 280.00              3990.00           4270.
00
*  !    2    0.1                  80.00              2230.00           2310.
00
*       3    0.1                 110.00              2220.00           2330.
00
*  !    4    0.5                  80.00              2230.00           2310.
00
*       5    0.5                  80.00              2230.00           2310.
00
*
* Time used             : 0.551
* Nb of solutions found : 5
*
Best solution:
--------------------------------------------------------------------
DRYER                      start         end   earliness   tardiness
--------------------------------------------------------------------
TEAK03                         0          90        0.00       70.00
TEAK04                        90         180        0.00      420.00
TEAK05                       180         270       10.00        0.00
TEAK06                       270         360        0.00      560.00
PINE00                       360         430        0.00      810.00
TEAK07                       430         520       40.00        0.00
```

```
TEAK08                         520         610       30.00        0.00
PINE01                         610         680        0.00        0.00
PINE02                         680         750        0.00       90.00
TEAK09                         750         840        0.00      280.00

--------------------------------------------------------------------
tot_earliness          :         80.00
tot_tardiness          :       2230.00
total                  :       2310.00
--------------------------------------------------------------------
```

The solution is displayed by resource (DRYER) with activities on each resource listed in chronological order. The earliness and tardiness costs for each activity are listed along with the total summation of all costs.

Above the displayed solution is a list of all solutions found, the solution number (no), the computation time since starting to solve, and the total costs of each solution.

Model and solve using the API for C++

This section describes in detail how to model and solve a manufacturing problem using the PowerOps API for C++.

In this section

Overview
Introduces the problem to be solved.

General approach to C++ modeling for Plant PowerOps
Compares modeling in C++ to modeling with data schema tables.

Define the necessary C++ functions
Describes the C++ functions that you need to build in order to model the problem.

Build the C++ program
Builds the entire set of functions to model the calendar problem.

Overview

This section discusses how to create and solve a model using the Plant PowerOps API for C++. This is built around a problem that uses calendars, and includes the modeling of setup activities, breaks, mode costs, unperformed activities, and productivity. This problem is also modeled and solved using the Java™ API in *Using the PPO API for Java to model and solve*.

Once you have modeled the problem for Plant PowerOps in a cpp file, you can solve the problem by running the program and viewing the output. The sample C++ program discussed here (along with several others) is provided in *<PPOInstallDirectory>*`/examples/src/api05calendar.cpp` in your development environment.

General approach to C++ modeling for Plant PowerOps

In general, there is a correspondence between modeling data in tables and modeling in C++, so you can follow a similar pattern as used for modeling with tables.

- ♦ Define the schema tables, their required fields, and any optional fields you are using for each table.
- ♦ Then define and load the data for each table.

Next you create a `main` program that:

- ♦ creates an empty `IloMSModel` object,
- ♦ loads the `IloMSModel` object with the problem data,
- ♦ calls the Plant PowerOps engine to solve the problem, and
- ♦ generates output and handles exceptions.

Note: To make porting easier from platform to platform, Plant PowerOps isolates characteristics that vary from system to system. For that reason, use the following names for basic types in C++:

- `IloInt` represents signed long integers
- `IloAny` represents pointers (void*)
- `IloNum` represents double precision floating-point values
- `IloBool` represents Boolean values: `IloTrue` and `IloFalse`

Define the necessary C++ functions

Every Plant PowerOps C++ program starts by declaring the Plant PowerOps include file, support for i/o stream operations, and `ILOSTBEGIN` as shown below:

```
#include <ilplant/schedengine.h>
#include <strstream>
ILOSTLBEGIN
```

After the initial declaration, this sample program has the following primary routines:

- ♦ a `SetModel` function that:
  - sets the problem name, date origin, time unit and minimal start time.
- ♦ a `SetCalendars` function that:
  - represents the calendar data for the problem;
  - declares the calendar objects for the problem; and
  - creates the calendar intervals.
- ♦ a `SetSetupMatrices` function that:
  - creates the matrix you need for the resource setup; and
  - fills in the matrix with the time and cost for possible combinations of `FROM_STATE` and `TO_STATE` values.
- ♦ a `SetResources` function that:
  - declares the resource objects you need for the problem;
  - sets the name for these resources;
  - gets the setup matrix;
  - associates the setup matrix with the resource; and
  - sets the initial setup state of the resource.
- ♦ a `SetMaterials` function that:
  - represents the materials data for the problem;
  - declares the materials objects for the problem; and
  - sets the name for these materials.
- ♦ a `SetRecipes` function that:
  - represents the recipe data for the problem;
  - declares the recipe objects for the problem; and

    - sets the name for these recipes.

- a `SetActivities` function that:
    - represents the activity data for the problem;
    - gets the recipe associated with the activity prototype;
    - declares the activity prototypes for the problem;
    - sets the name for these activities;
    - sets the setup state required by the activity;
    - sets the activity performed status; and
    - creates an activity identifier for easier retrieval.
- a `SetModes` function that:
    - represents the mode data for the activities in this problem;
    - gets the identifier of the activity prototype;
    - declares the mode objects for the problem;
    - gets the resource associated with the mode;
    - gets the calendar associated with the mode;
    - sets the variable processing time for the mode;
    - sets the name for the mode;
    - sets a cost for the mode;
    - sets an unperformed cost for the mode;
    - sets the calendar for the mode;
    - sets the maximum break duration for the mode;
    - sets the shift breakable value for the mode; and
    - assigns a resource constraint to the mode to indicate the required resource, the required capacity, and whether it is the primary resource for this activity in this mode.
- a `SetMaterialProductions` function that:
    - gets the recipes;
    - gets the materials associated with each recipe;
    - gets the activity associated with each recipe; and
    - links these three objects, assigning a default quantity to each recipe/materials/activity object.
- a `SetDemands` function that:

- represents the demand data (and nondelivery costs) for the problem;
- gets the materials associated with each demand;
- declares the demand objects for the problem; and
- sets the name and nondelivery costs for these demands.

♦ a `SetDueDates` function that:

- represents the due date data (and earliness/tardiness variable costs) for the problem;
- declares the due date objects for the problem;
- gets the demand associated with each due date;
- gets the due date assigned to each demand; and
- assigns an earliness variable cost and a tardiness variable cost to the activity.

♦ a `SetProductionOrders` function that:

- represents the production order data for the problem;
- gets the recipe associated with each production order;
- declares the production order objects and names for the problem.

♦ a `SetProductionToDemandArcs` function that:

- gets the production orders;
- gets the demand associated with each production order;
- gets the quantity associated with each demand; and
- links these objects so that the production order responds to the demand.

♦ a `SetCriterionWeights` function that:

- sets the global weight of all criteria used in the problem.

♦ a `main` program as described previously.

Build the C++ program

The following numbered steps “walk you through” the process of writing the C++ functions described previously. Refer to the *Plant PowerOps C++ Reference Manual* for more information about the methods described here.

In your C++ development environment, open the example file <*YourInstallDirectory*>/ `examples/src/api05calendar.cpp` and follow along in that sample as each step is discussed.

Step 1 Create the SetModel function

The first step creates a `SetModel` function:

1. Use the `setName` method to set the model name.
2. Use the `setIntDateOrigin` method to set the date origin (31*24*60*60, or 2678400 seconds since January 1, 2001, 00:00) for the problem.
3. Use the `setTimeUnit` method to set a time unit of minutes (60 seconds) for the problem.
4. Use the `setStartMin` method to set a start time of zero for the problem.

The completed `SetModel` function for this problem is as follows:

```
void SetModel(IloMSModel model)
{

  model.setName("EXAMPLE05");

  model.setIntDateOrigin(31*24*60*60);

  model.setTimeUnit(60);

  model.setStartMin(0);
}
```

Step 2 Create the SetCalendars function

This step creates a `SetCalendars` function for the problem to define periodic breaks.

1. Start by representing the incoming calendar data as an array.
2. Use the `newCalendar` method to create a calendar object for the problem.
3. Use `setBreak`, `setEndOfShift`, and other functions to declare the properties of the calendar intervals.

The completed `SetCalendars` function for this problem is as follows:

```
void SetCalendars(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfCalendars = 2;
  const char* names [2] = {"PINE_CALENDAR", "TEAK_CALENDAR"};
  IloInt numberOfCalendarIntervals = 8;
  IloInt calendarIndices [8] = {0, 0, 0, 1, 1, 1, 1, 1};
  IloInt startTimes [8] = {  0, 480, 960,
                             0, 480, 720, 780,  960};
  IloInt endTimes   [8] = {480, 960, 1440,
                           480, 720, 780, 960, 1440};
  IloBool breaks    [8] = {IloFalse, IloFalse, IloFalse,
                           IloFalse, IloFalse, IloTrue,  IloFalse, IloFalse};

  IloBool shiftEnds [8] = {IloTrue,  IloTrue,  IloTrue,
                           IloTrue,  IloFalse, IloFalse, IloTrue,  IloTrue};
  IloNum productivities [8] = {0.875, 1.000, 0.875,
                               0.750, 1.000, 1.000, 1.000, 1.000};
  // Create calendars.
  for (IloInt c = 0; c < numberOfCalendars; c++) {
    // Create the calendar.
    IloMSCalendar calendar = model.newCalendar();
    // Set the calendar name.
    calendar.setName(names[c]);
  }
  // Create calendar intervals.
  for (IloInt i = 0; i < numberOfCalendarIntervals; i++) {
    // Get the calendar.
    IloMSCalendar calendar = model.getCalendar(calendarIndices[i]);
    // Create the calendar interval.
    IloMSCalendarInterval calendarInterval =
      model.newCalendarInterval(calendar, startTimes[i], endTimes[i]);
    // Set its break indicator.
    calendarInterval.setBreak(breaks[i]);
    // Set its end of shift indicator.
    calendarInterval.setEndOfShift(shiftEnds[i]);
    // Set its productivity.
    calendarInterval.setProductivity(productivities[i]);
    // Set its periodicity and limit the period during which it applies.
    calendarInterval.setPeriodicity(1440);
    calendarInterval.setPeriodStartTime(0);
    calendarInterval.setPeriodEndTime(7200);
  }
}
```

Step 3 Create the SetSetupMatrices function

The next step creates a `SetSetupMatrices` function for this problem to define the possible relationships between the setup states. You also use this table to define a `SETUP_TIME` (duration, in `TIME_UNIT`s, the first numeric argument) for the setup state and a `SETUP_COST` (the second numeric argument) factor associated with it.

The completed `SetSetupMatrices` function for this problem is as follows:

```
void SetSetupMatrices(IloMSModel model)
{
  // Create the setup matrix.
  IloMSSetupMatrix matrix = model.newSetupMatrix();
  // Fill the values in the matrix.
  matrix.setSetup("PINE", "PINE", 0, 0);
  matrix.setSetup("PINE", "TEAK", 5, 50);
  matrix.setSetup("TEAK", "PINE", 5, 50);
  matrix.setSetup("TEAK", "TEAK", 0, 0);
}
```

Step 4 Create the SetResources function

The next step creates a `SetResources` function for this problem:

1. Start by using the `newResource` member function to create the resource. The argument is `Capacity`, which in this example is `1`.
2. Next, use the `setName` method to set a name for this problem's resource.
3. Use the `getSetupMatrix` method to get the matrix. The argument is the matrix being retrieved.
4. Use the `setSetupMatrix` method to associate the matrix with the resource.
5. Use the `setInitialSetupState` method to set the initial resource state.

The completed `SetResources` function for this problem is as follows:

```
void SetResources(IloMSModel model)
{
  // Create the resource.
  IloMSResource resource = model.newResource(1);
  // Set the resource name.
  resource.setName("DRYER");
  // Get the setup matrix (the unique setup matrix in the problem).
  IloMSSetupMatrix matrix = model.getSetupMatrix(0);
  // Associate the setup matrix with the resource.
  resource.setSetupMatrix(matrix);
  // Set the initial setup of the resource to PINE.
  resource.setInitialSetupState("PINE");
}
```

Step 5 Create the SetMaterials function

This step creates a `SetMaterials` function for the problem:

1. Start by representing the incoming materials data as an array.
2. In a `for` loop, use the `newMaterial` member function to create the material objects.
3. In the same `for` loop, use the `setName` method to assign a name from the data array to each of the materials.

The completed `SetMaterials` function for this problem is as follows:

```
void SetMaterials(IloMSModel model)
{

  IloInt numberOfMaterials = 2;
  char* names [2] = {"PINE_CRADLE", "TEAK_CRADLE"};

  for (IloInt i = 0; i < numberOfMaterials; i++) {

    IloMSMaterial material = model.newMaterial();

    material.setName(names[i]);
  }
}
```

Step 6 Create the SetRecipes function

This step creates a `SetRecipes` function for the problem:

1. Start by representing the incoming recipe data as an array.
2. In a `for` loop, use the `newRecipe` member function to create the recipe objects.
3. In the same `for` loop, use the `setName` method to assign a name from the data array to each of the recipes.

The completed `SetRecipes` function for this problem is as follows:

```
void SetRecipes(IloMSModel model)
{

  IloInt numberOfRecipes = 2;
  char* names [2] = {"PINE_RECIPE", "TEAK_RECIPE"};

  for (IloInt i = 0; i < numberOfRecipes; i++) {

    IloMSRecipe recipe = model.newRecipe();

    recipe.setName(names[i]);
  }
}
```

Step 7 Create the SetActivities function

You use this function to define two *types* of activities, or activity prototypes. You also associate a `RECIPE_ID`, a `SETUP_STATE`, and a `PERFORMED_STATUS` with each activity prototype. You also use the `setIdentifier` method to assign an identifier to the activity for easier retrieval in the `SetModes` function. The following lines create a `SetActivities` function for this problem:

1. Start by representing the incoming activity data as an array.

2. In a `for` loop, use the `getRecipe` method to get the recipe for each activity. The argument is the recipe being retrieved.
3. In the same `for` loop, use the `newActivityPrototype` member function to create the activity objects.
4. Still in the same `for` loop, use the `setName` method to assign a name from the data array to each of the activities.
5. Use the `setSetupState` method to assign a setup state from the data array to the activity.
6. Use the `setPerformedStatus` method to assign a performed status from the data array to each of the activities.
7. Finally, use the `setIdentifier` method to associate an identifier with each activity for easier retrieval.

The completed `SetActivities` function for this problem is as follows:

```
void SetActivities(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfActivityPrototypes = 2;
  const char* names [2] = {"PINE_DRYING", "TEAK_DRYING"};
  const char* states[2] = {"PINE", "TEAK"};
  IloMSPerformedStatus statuses [2];
  statuses[0] = IloMSPerformedOrUnperformed;
  statuses[1] = IloMSPerformed;
  // Create activities.
  for (IloInt i = 0; i < numberOfActivityPrototypes; i++) {
    // Get the ith recipe.
    IloMSRecipe recipe = model.getRecipe(i);
    // Create a new activity.
    IloMSActivity activity = model.newActivityPrototype(recipe);
    // Set the activity name.
    activity.setName(names[i]);
    // Set the activity setup state.
    activity.setSetupState(states[i]);
    // Set the activity performance status.
    activity.setPerformedStatus(statuses[i]);
    // Set the activity identifier (for easier retrieval).
    activity.setIdentifier(names[i]);
  }
}
```

Step 8 Create the SetModes function

You link each activity prototype with its corresponding mode, and assign a `COST` and an `UNPERFORMED_COST` to each mode. This step creates a `SetModes` function for this problem:

1. Start by representing the incoming mode data as an array.
2. In a `for` loop, use the `activityIds` method to retrieve the identifiers you assigned to each activity prototype in the previous function.

3. Use the `getActivityPrototypeByIdentifier` method to get each activity by identifier.
4. Use the `getResource` method to get the resource object being associated with this mode. The argument is the resource being retrieved.
5. Use the `getCalendar` method to get the calendar being associated with this mode. The argument is the calendar being retrieved.
6. Still in the loop, use the `newMode` member function to create the mode object. Its argument is the current `activity` in the loop.
7. Use the `setVariableProcessingTime` method to set the variable processing time for this mode. The argument is the time, from the data array.
8. Use the `setName` method to assign a name to the mode.
9. Use the `setVariableCost` method to assign a cost to the mode.
10. Use the `setUnperformedCost` method to assign an unperformed cost to the activity in this mode.
11. Use the `setCalendar` method to associate a calendar with the activity in this mode.
12. Use the `setBreakDurationMax` method to set the maximum break duration for the activity in this mode.
13. Use the `setShiftBreakable` method to define whether the activity can be interrupted by a break in this mode. The argument is a boolean.
14. Use the `newResourceConstraint` member function to indicate the resource's capacity and whether it is the primary resource for this activity in this mode. The first argument is the `mode` to which the `resource` is being assigned. The third argument is the `RequiredCapacity` of the `resource` in this `mode`. The fourth argument is a boolean that indicates whether the `resource` is a primary resource for the activity in this mode.

The completed `SetModes` function for this problem is as follows:

```
void SetModes(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfModes = 3;
  const char* activityIds [3] = {"PINE_DRYING", "TEAK_DRYING", "TEAK_DRYING"}
;
  IloInt processingTimes [3] = {70, 90, 60};
  const char* names [3] = {"PINE_MODE00", "TEAK_MODE00", "TEAK_MODE01"};
  IloInt modeCosts [3] = {100, 100, 200};
  IloInt unperformedCosts [3] = {100, 0, 0};
  IloInt calendarIndices [3] = {0, 1, 1};
  // Create modes.
  for (IloInt i = 0; i < numberOfModes; i++) {
    // Get the activity identifier.
    const char* id = activityIds[i];
    // Get the activity.
    IloMSActivity activity = model.getActivityPrototypeByIdentifier(id);
    // Get the resource (the unique resource in the problem).
    IloMSResource resource = model.getResource(0);
    // Get the calendar.
    IloMSCalendar calendar = model.getCalendar(calendarIndices[i]);
    // Create a new mode for the activity.
    IloMSMode mode = model.newMode(activity);
    // Set the variable processing time.
    mode.setVariableProcessingTime(processingTimes[i]);
    // Set the mode name.
    mode.setName(names[i]);
    // Set the processing cost.
    mode.setVariableCost(modeCosts[i]);
    // Set the unperformed cost.
    mode.setUnperformedCost(unperformedCosts[i]);
    // Set the calendar.
    mode.setCalendar(calendar);
    mode.setBreakDurationMax(0);
    mode.setShiftBreakable(IloFalse);
    // Create a new resource constraint.
    model.newResourceConstraint(mode, resource, 1, IloTrue);
  }
}
```

Step 9 Create the SetMaterialProductions function

The following lines create the function which *links* the `recipe` and the `materials` that it produces with each of the activity prototypes. You also use this table to define a default `Quantity` for the materials produced from this recipe.

1. To populate a `for` loop, use the `getNumberOfRecipes` method to retrieve the recipes created in the `SetRecipes` function.
2. In the `for` loop, use the `getRecipe` method to get each recipe object being associated with its materials. The argument is the recipe being retrieved.
3. In the same `for` loop, use the `getMaterial` method to get each material object being associated with its corresponding recipe. The argument is the material retrieved.
4. Still in the `for` loop, use the `getActivityPrototype` method to get each activity prototype being associated with its corresponding recipe and materials. As a result, due dates on demands for the material will automatically be translated into due dates on the appropriate activity instances. The argument is the activity prototype in the loop.
5. Finally, use the `newMaterialProduction` member function to state the quantity of material that the activity creates.

The completed `SetRecipeProducedMaterials` function for this problem is as follows:

```
void SetMaterialProductions(IloMSModel model)
{
  IloInt numberOfRecipes = model.getNumberOfRecipes();
  for (IloInt i = 0; i < numberOfRecipes; i++) {
    // Get the ith recipe.
    IloMSRecipe recipe = model.getRecipe(i);
    // Get the ith material.
    IloMSMaterial material = model.getMaterial(i);
    // Get the producing activity, i.e., the unique activity associated
    // with the recipe.
    IloMSActivity activity = recipe.getActivityPrototype(0);
    // State that the given activity produces the given material.
    model.newMaterialProduction(activity, material, 1.0);
  }
}
```

Step 10 Create the SetDemands function

The `Demand` object associates a `Demand` with its `Material`s. You can also specify the `Quantity` specified by the customer demand. The following steps create a `SetDemands` function for this problem:

1. Start by representing the incoming data as an array, including the nondelivery costs for the demands.
2. In a `for` loop, use the `getMaterial` method to get each material object being associated with a demand. The argument is the material being retrieved.
3. In the same `for` loop, use the `newDemand` member function to create the demand object. Its arguments are the current `Material` in the loop and a `Quantity` of `1.0` to indicate that this demand is for a single unit of the `Material`.
4. Use the `setName` method to associate a name with the demand object. Values for these names come from the name array.
5. The final instruction in the loop associates the nondelivery cost to the demand.

The completed `SetDemands` function for this problem is as follows:

```
void SetDemands(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfDemands = 10;
  IloInt materialIndices [10] = {0, 0, 0,
                                 1, 1, 1, 1,
                                 1, 1, 1};
  const char* names [10] = {"DEMAND00", "DEMAND01", "DEMAND02",
                            "DEMAND03", "DEMAND04", "DEMAND05", "DEMAND06",
                            "DEMAND07", "DEMAND08", "DEMAND09"};
  IloNum nonDeliveryVariableCosts [10] = {30000, 30000, 30000,
                                          70000, 70000, 70000, 70000,
                                          70000, 70000, 70000};
  // Create demands.
  for (IloInt i = 0; i < numberOfDemands; i++) {
    // Get the material.
    IloMSMaterial material = model.getMaterial(materialIndices[i]);
    // Create a new demand for one unit of the material.
    IloMSDemand demand = model.newDemand(material, 1.0);
    // Set the demand name.
    demand.setName(names[i]);
    demand.setNonDeliveryVariableCost(nonDeliveryVariableCosts[i]);
  }
}
```

Step 11 Create the SetDueDates function

The following steps create a `SetDueDates` function for this problem.

1. Start by representing the incoming due date data as an array, with earliness and tardiness variable costs.
2. In a `for` loop, use the `getDemand` method to get each demand object being associated with a due date. The argument is the demand being retrieved.
3. In the same `for` loop, use the `newDueDate` member function to create the due date object. Its arguments are the current `Demand` in the loop and the `DueTime` for that demand, both from the data array.
4. Finally, still in the `for` loop, use the `setEarlinessVariableCost` method to associate an earliness variable cost with the activity and the `setTardinessVariableCost` method to give it a tardiness variable cost. Values for these costs come from the data array.

The completed `SetDueDates` function for this problem is as follows:

```
void SetDueDates(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfDemands = 10;
  IloInt dueTimes               [10] = {160, 680, 720,
                                         80, 120, 280, 280,
                                        560, 640, 800};
  IloNum earlinessVariableCosts [10] = {1.0, 1.0, 1.0,
                                        1.0, 1.0, 1.0, 1.0,
                                        1.0, 1.0, 1.0};
  IloNum tardinessVariableCosts [10] = {3.0, 3.0, 3.0,
                                        7.0, 7.0, 7.0, 7.0,
                                        7.0, 7.0, 7.0};
  // Create due dates.
  for (IloInt i = 0; i < numberOfDemands; i++) {
    // Get the ith demand.
    IloMSDemand demand = model.getDemand(i);
    // Create a new due date for the demand.
    IloMSDueDate dd = model.newDueDate(demand, dueTimes[i]);
    // Set its earliness and tardiness variable costs.
    dd.setEarlinessVariableCost(earlinessVariableCosts[i]);
    dd.setTardinessVariableCost(tardinessVariableCosts[i]);
  }
}
```

Step 12 Create the SetProductionOrders function

This step creates the `SetProductionOrders` function:

1. Start by representing the incoming production order data as arrays.

2. Retrieve the current batching solution.
3. In a `for` loop, use the `getRecipe` method to get each recipe object being associated with a production order. The argument is the recipe being retrieved.
4. In the same `for` loop, use the `newProductionOrder` member function to create the production order object. Its argument is the current `Recipe` in the loop, from the data array.
5. Finally, still in the `for` loop, use the `setName` method to associate a name with the production order. Values for these names come from the data array.

The completed `SetProductionOrders` function for this problem is as follows:

```
void SetProductionOrders(IloMSModel model)
{
  // Represent problem data.
  IloInt numberOfOrders = 10;
  IloInt recipeIndices [10] = {0, 0, 0,
                               1, 1, 1, 1,
                               1, 1, 1};
  const char* names [10] = {"ORDER00", "ORDER01", "ORDER02",
                            "ORDER03", "ORDER04", "ORDER05", "ORDER06",
                            "ORDER07", "ORDER08", "ORDER09"};
  // Create production orders.
  IloMSBatchingSolution bs = model.getCurrentBatchingSolution();
  for (IloInt i = 0; i < numberOfOrders; i++) {
    // Get the recipe.
    IloMSRecipe recipe = model.getRecipe(recipeIndices[i]);
    // Create a new production order with the given recipe.
    IloMSProductionOrder order = bs.newProductionOrder(recipe);
    // Set the production order name.
    order.setName(names[i]);
  }
}
```

Step 13 Create the SetProductionToDemandArcs function

The following lines create the `SetProductionToDemandArcs` function, which *links* the `ProductionOrder` with the `Demand` that drives it.

1. To populate a `for` loop, use the `getNumberOfProductionOrders` method to retrieve the production orders created in the `SetProductionOrders` function.
2. In the `for` loop, use the `getProductionOrder` method to get each production order object being associated with its demand. The argument is the production order being retrieved.
3. In the same `for` loop, use the `getDemand` method to get each demand object being associated with its corresponding production order.
4. Still in the `for` loop, use the `getQuantity` method to get the quantity being associated with its corresponding production order and materials.

5. Finally, use the `newProdToDemandArc` member function to create the link. Its arguments are the current `Demand`, `order`, and `Quantity` in the loop.

The completed `SetProductionToDemandArcs` function for this problem is as follows:

```
void SetProductionToDemandArcs(IloMSModel model)
{
  IloInt numberOfOrders = model.getNumberOfProductionOrders();
  for (IloInt i = 0; i < numberOfOrders; i++) {
    // Get the ith order.
    IloMSProductionOrder order = model.getProductionOrder(i);
    // Get the ith demand.
    IloMSDemand demand = model.getDemand(i);
    // Get the quantity.
    IloNum quantity = demand.getQuantity();
    // State that the production order responds to the demand.
    model.newProdToDemandArc(demand.getMaterial(), quantity, order, demand);
  }
}
```

Step 14 Create the SetCriterionWeights function

The next step creates a `SetCriterionWeights` function for this problem. The `setWeight` method is used to set the cost of the criteria for the problem.

The completed `SetCriterionWeights` function for this problem is as follows:

```
void SetCriterionWeights(IloMSModel model)
{
  // Set the global weight of the total nondelivery criterion.
  model.setWeight("TotalNonDeliveryCost", 1.0);
 // Set the global weight of the total processing cost criterion.
  model.setWeight("TotalProcessingCost", 1.0);
  // Set the global weight of the total setup cost criterion.
  model.setWeight("TotalSetupCost", 1.0);
  // Set the global weight of the total earliness criterion.
  model.setWeight("TotalEarlinessCost", 1.0);
  // Set the global weight of the total tardiness criterion.
  model.setWeight("TotalTardinessCost", 1.0);
  // Set the global weight of the total unperformed cost criterion.
  model.setWeight("TotalUnperformedCost", 1.0);
}
```

Step 15 Create the main program

To complete the program, create an empty `IloMSModel` object, load it using all the functions you have just created, create a Plant PowerOps engine, and call the `solve` method. In the end, a call to the `model.end()` method frees all the memory allocated in the given `IloMSModel`.

```
class Arguments {
```

```
public:
  Arguments(int argc, char** argv);
  ~Arguments() {}
  IloInt getTraceLevel() const {return _traceLevel;}
private:
  IloInt _traceLevel;
};

Arguments::Arguments(int argc, char** argv)
:_traceLevel(1)
{
  for (int argIndex = 1; argIndex < argc; argIndex++) {
    if (!strcmp(argv[argIndex], "-trace")) {
      argIndex++;
      _traceLevel = atoi(argv[argIndex]);
    }
  }
}

int main(int argc, char** argv)
{
  try {
    Arguments args(argc, argv);
    // Create an empty IloMSModel object.
    IloMSModel model;
    model.setTraceLevel(args.getTraceLevel());
    // Set the manufacturing scheduling problem data.
    SetModel(model);
    SetCalendars(model);
    SetSetupMatrices(model);
    SetResources(model);
    SetMaterials(model);
    SetRecipes(model);
    SetActivities(model);
    SetModes(model);
    SetMaterialProductions(model);
    SetDemands(model);
    SetDueDates(model);
    SetProductionOrders(model);
    SetProductionToDemandArcs(model);
    SetCriterionWeights(model);
    // Create a scheduling engine.
    IloMSSchedulingEngine engine(model);
    // Solve the problem, print the results, clean up the IloMSModel
    // object, and exit.
    if (engine.solve()) {
      cout << "Best solution:" << endl;
      cout << model.getCurrentSchedulingSolution() << endl;
      model.end();
      return 0;
    }
    else {
      cout << "No solution" << endl;
      model.end();
      return -1;
    }
  }
  catch (IloMSException& exception) {
    cout << exception.getMessage() << endl;
    return -1;
  }
  catch (...) {
    cout << "Unknown error." << endl;
    return -1;
  }
}
```

This is the final step in creating the CPP file for this example.

Step 16 Compile and run the program

Use the sample provided in *InstallDirectory*/`examples/src/api05calendar.cpp` in your development environment to compile and run the program. Running this program provides output similar to the following, which can be used to assess the generated plan.

```
* Start solving
*
* Time limit     : 10
*
*     no  time  tot_mode_cost  tot_setup_cost  tot_earliness  tot_tardiness
tot_unperf_cost       total
*
*  !   1   0.1        1000.00          200.00         209.00       10371.00
           0.00    11780.00
*  !   2   0.1        1100.00          300.00           0.00        7964.00
           0.00     9364.00
*  !   3   0.1        1300.00          150.00         718.00        3961.00
         200.00     6329.00
*  !   4   0.2        1300.00          100.00         728.00        2211.00
         200.00     4539.00
*  !   5   0.2        1400.00          250.00         355.00         847.00
         100.00     2952.00
*      6   0.2        1400.00          250.00         355.00         847.00
         100.00     2952.00
*      7   0.2        1400.00          150.00        1059.00         742.00
         200.00     3551.00
*      8   0.3        1400.00          150.00        1059.00         742.00
         200.00     3551.00
*  !   9   0.3        1500.00          150.00         445.00         707.00
         100.00     2902.00
*     10   0.3        1500.00          150.00         445.00         707.00
         100.00     2902.00
*  !  11   0.5        1400.00          150.00         335.00         777.00
         100.00     2762.00
*     12   1.7        1400.00          150.00         335.00         777.00
         100.00     2762.00
*     13   7.0        1400.00          150.00         335.00         777.00
```

```
         100.00     2762.00
*     14   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     15   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     16   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*  !  17   7.0       1400.00         150.00         223.00         777.00
         100.00     2650.00
*     18   7.1       1400.00         150.00         223.00         777.00
         100.00     2650.00
*     19   7.3       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     20   7.3       1400.00         150.00         215.00         848.00
         100.00     2713.00
*     21   7.3       1400.00         150.00         215.00         848.00
         100.00     2713.00
*
* Time used            : 7.48
* Nb of solutions found : 21
*
Best solution:
-----------------------------------------------------------------------------
-------------------------
DRYER                          start        end mode  mode/setup   earliness
   tardiness unperformed
-----------------------------------------------------------------------------
-------------------------
ORDER03_TEAK_DRYING_setup          0          7    0          50         0.
00        0.00          0
ORDER03_TEAK_DRYING                7         87    1         200         0.
00       49.00          0
ORDER00_PINE_DRYING_setup         79         79    0           0         0.
00        0.00          0
ORDER00_PINE_DRYING               79        160    0         100         0.
00        0.00        100
ORDER04_TEAK_DRYING_setup         87         87    0           0         0.
00        0.00          0
ORDER04_TEAK_DRYING               87        167    1         200         0.
00      329.00          0
ORDER05_TEAK_DRYING_setup        167        167    0           0         0.
00        0.00          0
ORDER05_TEAK_DRYING              167        247    1         200        33.
00        0.00          0
ORDER06_TEAK_DRYING_setup        247        247    0           0         0.
00        0.00          0
ORDER06_TEAK_DRYING              247        327    1         200         0.
00      329.00          0
ORDER07_TEAK_DRYING_setup        360        360    0           0         0.
00        0.00          0
ORDER07_TEAK_DRYING              360        480    0         100        80.
00        0.00          0
ORDER08_TEAK_DRYING_setup        480        480    0           0         0.
00        0.00          0
```

```
ORDER08_TEAK_DRYING              480         570    0         100       70.
00        0.00           0
ORDER01_PINE_DRYING_setup        570         575    0          50        0.
00        0.00           0
ORDER01_PINE_DRYING              575         645    0         100       35.
00        0.00           0
ORDER02_PINE_DRYING_setup        645         645    0           0        0.
00        0.00           0
ORDER02_PINE_DRYING              645         715    0         100        5.
00        0.00           0
ORDER09_TEAK_DRYING_setup        715         720    0          50        0.
00        0.00           0
ORDER09_TEAK_DRYING              720         810    0         100        0.
00       70.00           0

-----------------------------------------------------------------------------
-------------------------
tot_mode_cost        :      1400.00
tot_setup_cost       :       150.00
tot_earliness        :       223.00
tot_tardiness        :       777.00
tot_unperf_cost      :       100.00
total                :      2650.00
-------------------------------------------------------------------------
```

Using the PPO API for Java to model and solve

The first section describes how to use Java™ to solve a problem model built with csv database files. The second section describes how to use the Java API to model and solve a scheduling problem that uses calendars.

In this section

Solving using csv files with Java
Describes how to use Java to solve your manufacturing problem data model.

Model and solve using the API for Java
This section discusses how to create the model for the calendar problem using the Plant PowerOps API for Java™ . This is built around a problem that uses calendars, and includes the modeling of setup activities, breaks, mode costs, unperformed activities, and productivity. This problem is also modeled and solved using the C++ API in *Using the PPO API for C++ to model and solve*. The sample Java program described here is provided in *<PPOInstallDirectory>*`/examples/src/api05calendar.java` in your development environment.

Solving using csv files with Java

A model built with csv tables can be solved using a Java™ program, as an alternative to solving in the GUI. A sample Java program `csvsch.java` is provided (located at *<PPOInstallDirectory>*`/examples/src/csvsch.java`) which you can use to optimize csv data files.

Compile `csvrun.java`, and use the resulting executable `csvsch.exe` to start optimization of a csv data file, such as `CourseLesson1_basic.csv`. Details regarding syntax on different platforms can be found in *<PPOInstallDirectory>*`/examples/src/csvsch.java`

Model and solve using the API for Java

This section discusses how to create the model for the calendar problem using the Plant PowerOps API for Java™ . This is built around a problem that uses calendars, and includes the modeling of setup activities, breaks, mode costs, unperformed activities, and productivity. This problem is also modeled and solved using the C++ API in *Using the PPO API for C++ to model and solve*. The sample Java program described here is provided in *<PPOInstallDirectory>*`/examples/src/api05calendar.java` in your development environment.

In this section

Differences between C++ and Java
Describes a few differences between modeling in the PPO C++ API versus the Java API.

Define the program Java functions
Lists all the Java functions that must be populated in order to model the calendar problem.

Build the Java program
Creates the Java functions and the complete program to solve the calendar problem.

Differences between C++ and Java

The Plant PowerOps API for Java™ is similar to the C++ API, so it is beneficial to read the previous chapter on C++. However, there are a number of significant differences.

Entry Point

The whole code is wrapped into a class, followed by the first static function, called `SetModel`:

```
public class api05calendar
{
   static void SetModel(IloMSModel model)
   {
   ...
   }
 ...
}
```

In the `api05calendar.java` program, all the global functions of the C++ example (`api05calendar.cpp`) have been converted into static member functions of the `api05calendar` class.

Import Vs. include

The `include` of the C++ header file is replaced by an `import` directive:

```
import ilog.plant.*;
```

Construction and destruction of model

The Java memory allocated is freed by the call to `IloMSModel.end()`. Omitting this call leads to unrecoverable memory leaks.

In Java, the instantiation of a model is done using the static factory method `IloMSModel.newModel()`.

Exceptions

Exceptions thrown in C++ are instances of `IloMSException`; in Java these become `RuntimeException`.

Primitive types

The C++ primitive types `IloInt`, `IloNum`, and `char*` exist in Java as `int`, `double`, and `java.lang.String`.

Enumerated types

The C++ enums are defined as classes in java, with the types being `int`. The name of the constant is the name of the corresponding C++ `enum` value without the `IloMS` prefix, proceeded by the class name. For example, `IloMSCleaning` in C++ becomes `IloMSCleaningStatus.Cleaning` in Java.

For a full list of the classes available, see the *Plant PowerOps Java Reference Manual*.

Define the program Java functions

The development of this example follows the general procedure described in *Using the PPO API for C++ to model and solve*.

Every Plant PowerOps Java™ API program starts by importing the Plant PowerOps include file as shown below:

```
import ilog.plant.*;
```

Next this sample program has these primary routines:

- a `SetModel` function that:
  - sets the problem name, date origin, time unit and minimum start time.
- a `SetCalendars` function that:
  - creates the calendar and calendar intervals for the problem
  - sets the start times, end times, and periodicity of the calendar intervals, and
  - declares the breaks, shifts, and productivity values for the calendar intervals.
- a `SetSetupMatrices` function that:
  - creates the matrix you need for the resource setup, and
  - fills in the matrix with the setup times and costs to go from one state to another.
- a `SetResources` function that:
  - declares the resource objects and their names that are needed for the problem
  - associates the setup matrix with the resource, and
  - sets the initial setup state of the resource.
- a `SetMaterials` function that:
  - represents the materials data for the problem
  - declares the materials objects for the problem, and
  - sets the name for these materials.
- a `SetRecipes` function that:
  - represents the recipe data for the problem
  - declares the recipe objects for the problem, and
  - sets the name for these recipes.
- a `SetActivities` function that:

    - represents the activity data for the problem
    - gets the recipe associated with the activity prototype
    - declares the activity prototypes for the problem
    - sets the name for these activities
    - sets the setup state required by the activity
    - sets the activity performed status, and
    - creates an activity identifier for easier retrieval.
- ♦ a `SetModes` function that:
    - represents the mode data for the activities in this problem
    - gets the identifier of the activity prototype and then retrieves the activity prototype
    - declares the mode objects for the problem
    - gets the resource and calendar associated with the mode
    - sets the name and variable processing time for the mode
    - sets a fixed and an unperformed cost for the mode
    - sets the calendar for the mode
    - sets the maximum break duration and the shift breakable values for the mode, and
    - assigns a resource constraint to the mode to indicate the required resource, the required capacity, and whether it is the primary resource for this activity in this mode.
- ♦ a `SetMaterialProductions` function that:
    - gets the recipes
    - gets the materials associated with each recipe
    - gets the producing activity associated with each recipe, and
    - assigns a production quantity to each recipe/materials/activity object.
- ♦ a `SetDemands` function that:
    - represents the demand data for the problem
    - gets the materials associated with each demand
    - declares the demand objects for the problem, and
    - sets the name for these demands.
- ♦ a `SetDueDates` function that:
    - represents the due date data (and earliness/tardiness variable costs) for the problem

- declares the due date objects for the problem
- gets the demands and due dates associated with each other, and
- assigns an earliness variable cost and a tardiness variable cost to the activity.

♦ a `SetProductionOrders` function that:

- represents the production order data for the problem
- gets the recipe associated with each production order
- declares the production order objects for the problem, and
- sets the name for these production orders.

♦ a `SetProductionToDemandArcs` function that:

- gets the production orders
- gets the demand associated with each production order
- gets the quantity associated with each demand, and
- links these objects so that the production order responds to the demand.

♦ a `SetCriterionWeights` function that:

- sets the global weight of all criteria used in the problem

♦ a `main` program as described in the previous section.

Build the Java program

The following numbered steps “walk you through” the process of building the Java™ functions previously described. Refer to the *Plant PowerOps Java Reference Manual* for more information about the methods described here.

In your Java development environment, open the example file *InstallDirectory*`/examples/src/api05calendar.java` and follow along in that sample as each step is discussed.

Step 1 Create the SetModel function

The first step creates the `SetModel` function:

1. Use the `setName` method to set the model name.
2. Use the `setIntDateOrigin` method to set the date origin (`31*24*60*60`, or 2678400 seconds since January 1, 2001, 00:00) for the problem.
3. Use the `setTimeUnit` method to set a time unit of minutes (`60` seconds) for the problem.
4. Use the `setStartMin` method to set a start time of zero for the problem.

The completed `SetModel` function for this problem is as follows:

```
static void SetModel(IloMSModel model)
{
  model.setName("EXAMPLE05");
  model.setIntDateOrigin(31*24*60*60);
  model.setTimeUnit(60);
  model.setStartMin(0);
}
```

Step 2 Create the SetCalendars function

This step creates a `SetCalendars` function for the problem, used to define the intervals specifying breaks, shifts, and productivity.

1. Start by declaring the name and number of the calendars, the intervals, and setting up the counter `calendarIndices` used later to assign the data to the intervals.
2. Enter the interval data; the start and end times, breaks, shifts, and productivity.
3. Use the `newCalendar` method to create a calendar object for the problem.
4. Use the `newCalendarInterval` method to create the calendar interval objects for the problem.
5. Use `setBreak` to indicate if the interval is a break or not.
6. Use `setEndOfShift` to indicate if the interval is an end of shift or not.
7. Use `setProductivity` to set the productivity for the interval.

8. Use `setPeriodicity`, `setPeriodStartTime`, `setPeriodEndTime` to set the overall time period in which the intervals exist; this is the same for all intervals.

The completed `SetCalendars` function for this problem is as follows:

```
static void SetCalendars(IloMSModel model)
{
  int numberOfCalendars = 2;
  String names [] = {"PINE_CALENDAR", "TEAK_CALENDAR"};
  int numberOfCalendarIntervals = 8;
  int calendarIndices [] = {0, 0, 0, 1, 1, 1, 1, 1};
  int startTimes    [] = {  0, 480, 960,
                            0, 480, 720, 780,  960};
  int endTimes      [] = {480, 960, 1440,
                          480, 720, 780, 960, 1440};
  boolean breaks    [] = {false, false, false,
                          false, false, true,  false, false};
  boolean shiftEnds [] = {true,  true,  true,
                          true,  false, false, true,  true};
  double productivities [] = {0.875, 1.000, 0.875,
                              0.750, 1.000, 1.000, 1.000, 1.000};
  for (int c = 0; c < numberOfCalendars; c++)
  {
    IloMSCalendar calendar = model.newCalendar();
    calendar.setName(names[c]);
  }
  for (int i = 0; i < numberOfCalendarIntervals; i++)
  {
    IloMSCalendar calendar = model.getCalendar(calendarIndices[i]);
    IloMSCalendarInterval calendarInterval =
      model.newCalendarInterval(calendar, startTimes[i], endTimes[i]);
    calendarInterval.setBreak(breaks[i]);
    calendarInterval.setEndOfShift(shiftEnds[i]);
    calendarInterval.setProductivity(productivities[i]);
    calendarInterval.setPeriodicity(1440);
    calendarInterval.setPeriodStartTime(0);
    calendarInterval.setPeriodEndTime(7200);
  }
}
```

Step 3 Create the SetSetupMatrices function

The next step creates a `SetSetupMatrices` function for this problem to define the possible relationships (including setup times and costs) between the setup states.

1. Start by using the `newSetupMatrix` member function to create the matrix.
2. Use the `setSetup` method to declare the "from" state and the "to" state, then the setup time, setup cost, and whether a cleanup is required or not.

The completed `SetSetupMatrices` function for this problem is as follows:

```
static void SetSetupMatrices(IloMSModel model)
```

```
{
  IloMSSetupMatrix matrix = model.newSetupMatrix();
  matrix.setSetup("PINE", "PINE", 0, 0,  false);
  matrix.setSetup("PINE", "TEAK", 5, 50, false);
  matrix.setSetup("TEAK", "PINE", 5, 50, false);
  matrix.setSetup("TEAK", "TEAK", 0, 0,  false);
}
```

Step 4 Create the SetResources function

The next step creates a `SetResources` function for this problem:

1. Start by using the `newResource` member function to create the resource. The argument is `Capacity`, which in this example is `1`. This means the resource cannot be shared at any one particular time.
2. Next, use the `setName` method to set a name for this problem's resource.
3. Use the `getSetupMatrix` method to get the matrix. The argument is the matrix being retrieved.
4. Use the `setSetupMatrix` method to associate the matrix with the resource. This associates setup times and costs with the resource states.
5. Use the `setInitialSetupState` method to set an initial value of `PINE` (in other words, the resource `DRYER` starts with an initial state read for drying `PINE`).

The completed `SetResources` function for this problem is as follows:

```
static void SetResources(IloMSModel model)
{

  IloMSResource resource = model.newResource(1);

  resource.setName("DRYER");

  IloMSSetupMatrix matrix = model.getSetupMatrix(0);

  resource.setSetupMatrix(matrix);

  resource.setInitialSetupState("PINE");
}
```

Step 5 Create the SetMaterials function

This step creates a `SetMaterials` function for the problem:

1. Start by representing the incoming materials data as an array.
2. In a `for` loop, use the `newMaterial` member function to create the material objects.
3. In the same `for` loop, use the `setName` method to assign a name from the data array to each of the materials.

The completed `SetMaterials` function for this problem is as follows:

```
static void SetMaterials(IloMSModel model)
{

  int numberOfMaterials = 2;
  String names [] = {"PINE_CRADLE", "TEAK_CRADLE"};

  for (int i = 0; i < numberOfMaterials; i++) {

    IloMSMaterial material = model.newMaterial();

    material.setName(names[i]);
  }
}
```

Step 6 Create the SetRecipes function

This step creates a `SetRecipes` function for the problem:

1. Start by representing the incoming recipe data as an array.
2. In a `for` loop, use the `newRecipe` member function to create the recipe objects.
3. In the same `for` loop, use the `setName` method to assign a name from the data array to each of the recipes.

The completed `SetRecipes` function for this problem is as follows:

```
static void SetRecipes(IloMSModel model)
{

  int numberOfRecipes = 2;
  String names [] = {"PINE_RECIPE", "TEAK_RECIPE"};

  for (int i = 0; i < numberOfRecipes; i++) {

    IloMSRecipe recipe = model.newRecipe();

    recipe.setName(names[i]);
  }
}
```

Step 7 Create the SetActivities function

This step creates the `SetActivities` function, which is used to define two *types* of activities, or activity prototypes. You associate a recipe, a setup state, and a performed status with each activity prototype.

1. Start by representing the incoming activity data as an array.
2. In a `for` loop, use the `getRecipe` method to get the recipe for each activity. The argument is the recipe being retrieved.

3. In the same `for` loop, use the `newActivityPrototype` member function to create the activity objects.
4. Still in the same `for` loop, use the `setName` method to assign a name from the data array to each of the activities.
5. Use the `setSetupState` method to assign a setup state from the data array to the activity.
6. Use the `setPerformedStatus` method to assign a performed status from the data array to each of the activities.
7. Finally, use the `setIdentifier` method to associate an identifier with each activity for easier retrieval in the `SetModes` function.

The completed `SetActivities` function for this problem is as follows:

```
static void SetActivities(IloMSModel model)
{
  int numberOfActivityPrototypes = 2;
  String names [] = {"PINE_DRYING", "TEAK_DRYING"};
  String states[] = {"PINE","TEAK"};
  int statuses [] = { IloMSPerformedStatus.PerformedOrUnperformed,
                      IloMSPerformedStatus.Performed };
  for (int i = 0; i < numberOfActivityPrototypes; i++) {

    IloMSRecipe recipe = model.getRecipe(i);

    IloMSActivity activity = model.newActivityPrototype(recipe);

    activity.setName(names[i]);

    activity.setSetupState(states[i]);

    activity.setPerformedStatus(statuses[i]);

    activity.setIdentifier(names[i]);
  }
}
```

Step 8 Create the SetModes function

You must link each activity prototype with its corresponding mode, and assign a fixed and variable cost to each mode. This step creates a `SetModes` function for this problem:

1. Start by setting the data in an array.
2. In a `for` loop, use the `getActivityByIdentifier` method to get each activity by identifier (previously set in the `SetActivities` function).
3. Use the `getResource` method to get the resource object (to be) associated with this mode. The argument is the resource being retrieved.
4. Use the `getCalendar` method to get the calendar (to be) associated with this mode. The argument is the calendar being retrieved.

5. Still in the loop, use the `newMode` member function to create the mode object. Its arguments are the current `Activity` in the loop.
6. Use the `setVariableProcessingTime` method to set the variable processing time for this mode. The argument is the time, from the data array.
7. Use the `setName` method to assign a name to the mode.
8. Use the `setFixedCost` method to assign a cost to the mode.
9. Use the `setUnperformedCost` method to assign an unperformed cost to the activity in this mode.
10. Use the `setCalendar` method to associate a calendar with the activity in this mode.
11. Use the `setBreakDurationMax` method to set the maximum break duration for the activity in this mode.
12. Use the `setShiftBreakable` method to define whether the activity can be interrupted by a break in this mode.
13. Use the `newResourceConstraint` method to indicate the resource's capacity and whether it is the primary resource for this activity in this mode. The first argument is the `Mode` to which the resource is being assigned. The second argument is the `Resource`. The third argument is the `RequiredCapacity` of the resource in this mode. The fourth argument is a boolean that indicates whether the resource is a primary resource for the activity in this mode.

The completed `SetModes` function for this problem is as follows:

```
static void SetModes(IloMSModel model)
{
  int numberOfModes = 3;
  String activityIds [] = {"PINE_DRYING", "TEAK_DRYING", "TEAK_DRYING"};
  int processingTimes [] = {70, 90, 60};
  String names [] = {"PINE_MODE00", "TEAK_MODE00", "TEAK_MODE01"};
  double modeCosts [] = {100, 100, 200};
  double unperformedCosts [] = {100, 0, 0};
  int calendarIndices [] = {0, 1, 1};

  for (int i = 0; i < numberOfModes; i++) {

    String id = activityIds[i];

    IloMSAbstractActivity activity = model.getActivityByIdentifier(id);

    IloMSResource resource = model.getResource(0);

    IloMSCalendar calendar = model.getCalendar(calendarIndices[i]);

    IloMSMode mode = model.newMode(activity);

    mode.setVariableProcessingTime(processingTimes[i]);

    mode.setName(names[i]);
```

```
      mode.setFixedCost(modeCosts[i]);

      mode.setUnperformedCost(unperformedCosts[i]);

      mode.setCalendar(calendar);
      mode.setBreakDurationMax(0);
      mode.setShiftBreakable(false);

      model.newResourceConstraint(mode, resource, 1, true);
    }
  }
```

Step 9 Create the SetMaterialProductions function

The following lines create the `SetMaterialProductions` function, which *links* the `Recipe` and the `Materials` that it produces with each of the activity prototypes. You also use this table to define a default `Quantity` for the materials produced from this recipe.

1. To populate a `for` loop, use the `getNumberOfRecipes` method to retrieve the recipes created in the `SetRecipes` function.
2. In the `for` loop, use the `getRecipe` method to get each recipe object being associated with its materials. The argument is the recipe being retrieved.
3. In the same `for` loop, use the `getMaterial` method to get each material object being associated with its corresponding recipe. The argument is the material being retrieved.
4. Still in the `for` loop, use the `getActivityPrototype` method to get each activity prototype being associated with its corresponding recipe and materials. As a result, due dates on demands for the material will automatically be translated into due dates on the appropriate activity instances.
5. Finally, use the `newMaterialProduction` member function to create the link between activity, material, and amount produced.

The completed `SetMaterialProductions` function for this problem is as follows:

```
static void SetMaterialProductions(IloMSModel model)
  {
    int numberOfRecipes = model.getNumberOfRecipes();
    for (int i = 0; i < numberOfRecipes; i++) {

      IloMSRecipe recipe = model.getRecipe(i);

      IloMSMaterial material = model.getMaterial(i);

      IloMSActivity activity = recipe.getActivityPrototype(0);

      model.newMaterialProduction(activity, material, 1.0);
    }
  }
```

Step 10 Create the SetDemands function

The Demand object associates a demand with its Material. You can also specify the Quantity specified by the customer demand. The following steps create a SetDemands function for this problem:

1. Start by representing the incoming data, including the nondelivery costs, as an array.
2. In a for loop, use the getMaterial method to get each material object being associated with a demand. The argument is the material being retrieved.
3. In the same for loop, use the newDemand member function to create the demand object. Its arguments are the current Material in the loop and a Quantity of 1.0 to indicate that this demand is for a single unit of the Material.
4. Use the setName method to associate a name with the demand object. Values for these names come from the name array.
5. Close the for loop after setting the nondelivery variable costs.

The completed SetDemands function for this problem is as follows:

```
static void SetDemands(IloMSModel model)
{

  int numberOfDemands = 10;
  int materialIndices [] = {0, 0, 0,
                            1, 1, 1, 1,
                            1, 1, 1};
  String names [] = {"DEMAND00", "DEMAND01", "DEMAND02",
                     "DEMAND03", "DEMAND04", "DEMAND05", "DEMAND06",
                     "DEMAND07", "DEMAND08", "DEMAND09"};
  double nonDeliveryVariableCosts [] = {30000, 30000, 30000,
                                        70000, 70000, 70000, 70000,
                                        70000, 70000, 70000};

  for (int i = 0; i < numberOfDemands; i++) {

    IloMSMaterial material = model.getMaterial(materialIndices[i]);

    IloMSDemand demand = model.newDemand(material, 1.0);

    demand.setName(names[i]);
    demand.setNonDeliveryVariableCost(nonDeliveryVariableCosts[i]);

  }
}
```

Step 11 Create the SetDueDates function

The following steps create a SetDueDates function for this problem.

1. Start by representing the incoming due date data as an array.

2. In a `for` loop, use the `getDemand` method to get each demand object being associated with a due date. The argument is the demand being retrieved.
3. In the same `for` loop, use the `newDueDate` member function to create the due date object. Its arguments are the current `Demand` in the loop and the `DueTime` for that demand, both from the data array.
4. Finally, still in the `for` loop, use the `setEarlinessVariableCost` method to associate an earliness variable cost with the activity and the `setTardinessVariableCost` method to give it a tardiness variable cost. Values for these costs come from the data array.

The completed `SetDueDates` function for this problem is as follows:

```
static void SetDueDates(IloMSModel model)
{

  int numberOfDemands = 10;
  int dueTimes               [] = {160, 680, 720,
                                   80, 120, 280, 280,
                                   560, 640, 800};
  double earlinessVariableCosts [] = {1.0, 1.0, 1.0,
                                   1.0, 1.0, 1.0, 1.0,
                                   1.0, 1.0, 1.0};
  double tardinessVariableCosts [] = {3.0, 3.0, 3.0,
                                   7.0, 7.0, 7.0, 7.0,
                                   7.0, 7.0, 7.0};

  for (int i = 0; i < numberOfDemands; i++) {

    IloMSDemand demand = model.getDemand(i);

    IloMSDueDate dd = model.newDueDate(demand, dueTimes[i]);

    dd.setEarlinessVariableCost(earlinessVariableCosts[i]);
    dd.setTardinessVariableCost(tardinessVariableCosts[i]);
  }
}
```

Step 12 Create the SetProductionOrders function

This step creates a `SetProductionOrders` function for this problem:

1. Start by representing the incoming production order data as arrays.
2. Retrieve the batching solution, then in a `for` loop, use the `getRecipe` method to get each recipe object being associated with a production order. The argument is the recipe being retrieved.
3. In the same `for` loop, use the `newProductionOrder` member function to create the production order object. Its argument is the current `Recipe` in the loop, from the data array.
4. Finally, still in the `for` loop, use the `setName` method to associate a name with the production order. Values for these names come from the data array.

The completed `SetProductionOrders` function for this problem is as follows:

```
static void SetProductionOrders(IloMSModel model)
{

  int numberOfOrders = 10;
  int recipeIndices [] = {0, 0, 0,
                          1, 1, 1, 1,
                          1, 1, 1};
  String names [] = {"ORDER00", "ORDER01", "ORDER02",
                    "ORDER03", "ORDER04", "ORDER05", "ORDER06",
                    "ORDER07", "ORDER08", "ORDER09"};
  IloMSBatchingSolution bs = model.getCurrentBatchingSolution();

  for (int i = 0; i < numberOfOrders; i++) {

    IloMSRecipe recipe = model.getRecipe(recipeIndices[i]);

    IloMSProductionOrder order = bs.newProductionOrder(recipe);

    order.setName(names[i]);
  }
}
```

Step 13 Create the SetProductionToDemandArcs function

The following lines create the `SetProductionOrderDemands` function, which links the `ProductionOrder` with the `Demand` that drives it.

1. To populate a `for` loop, use the `getNumberOfProductionOrders` method to retrieve the production orders created in the `SetProductionOrders` function.
2. In the `for` loop, use the `getProductionOrder` method to get each production order object being associated with its demand. The argument is the production order being retrieved.
3. In the same `for` loop, use the `getDemand` method to get each demand object being associated with its corresponding production order. The argument is the demand being retrieved.
4. Still in the `for` loop, use the `getQuantity` method to get the quantity being associated with its corresponding production order and materials.
5. Finally, use the `newProdToDemandArc` method to create a new material flow arc between the production order `order` and the `demand`. The first and second arguments specify the material and the `quantity`.

The completed `SetProductionToDemandArcs` function for this problem is as follows:

```
static void SetProductionToDemandArcs(IloMSModel model)
{
  int numberOfOrders = model.getNumberOfProductionOrders();
  for (int i = 0; i < numberOfOrders; i++) {
      IloMSProductionOrder order = model.getProductionOrder(i);

      IloMSDemand demand = model.getDemand(i);

      double quantity = demand.getQuantity();

     model.newProdToDemandArc(demand.getMaterial(), quantity, order, demand)
;
    }
  }
```

Step 14 Create the SetCriterionWeights function

The next step creates a `SetCriterionWeights` function for this problem. The `setWeight` method is used to sequentially set the weight of the five criteria in this problem to `1.0`.

The completed `SetCriterionWeights` function for this problem is as follows:

```
static void SetCriterionWeights(IloMSModel model)
{
  model.setWeight("TotalNonDeliveryCost", 1.0);
  model.setWeight("TotalProcessingCost", 1.0);
  model.setWeight("TotalSetupCost", 1.0);
  model.setWeight("TotalEarlinessCost", 1.0);
  model.setWeight("TotalTardinessCost", 1.0);
  model.setWeight("TotalUnperformedCost", 1.0);
}
```

Step 15 Create the main program

To complete the program, create an empty `IloMSModel` object, load it with the functions you have just created, create a Plant PowerOps engine, and call the `solve` method. In the end, a call to the `model.end()` method frees all the memory allocated in the given `IloMSModel`.

```
public static final void  main(String[] args)
{

  IloMSModel model = IloMSModel.newModel();

  SetModel(model);
  SetCalendars(model);
  SetSetupMatrices(model);
  SetResources(model);
  SetMaterials(model);
  SetRecipes(model);
  SetActivities(model);
  SetModes(model);
  SetMaterialProductions(model);
  SetDemands(model);
```

```
    SetDueDates(model);
    SetProductionOrders(model);
    SetProductionToDemandArcs(model);
    SetCriterionWeights(model);

    IloMSSchedulingEngine engine = IloMSSchedulingEngine.newSchedulingEngine
(model);

    engine.solve();

    model.end();
  }
}
```

This is the final step in creating the Java file for this example. Use the sample provided in *InstallDirectory*`/examples/src/api05calendar.java` in your development environment to complete the next steps and solve the problem.

Step 16 Compile and run the program

Compile and run the program to receive output similar to the following.

```
* Start solving
*
* Time limit     : 10
*
*     no  time  tot_mode_cost  tot_setup_cost  tot_earliness  tot_tardiness
tot_unperf_cost       total
*
*  !   1   0.1        1000.00          200.00         209.00       10371.00
           0.00    11780.00
*  !   2   0.1        1100.00          300.00           0.00        7964.00
           0.00     9364.00
*  !   3   0.1        1300.00          150.00         718.00        3961.00
         200.00     6329.00
*  !   4   0.2        1300.00          100.00         728.00        2211.00
         200.00     4539.00
*  !   5   0.2        1400.00          250.00         355.00         847.00
         100.00     2952.00
*      6   0.2        1400.00          250.00         355.00         847.00
         100.00     2952.00
*      7   0.2        1400.00          150.00        1059.00         742.00
         200.00     3551.00
*      8   0.3        1400.00          150.00        1059.00         742.00
         200.00     3551.00
*  !   9   0.3        1500.00          150.00         445.00         707.00
         100.00     2902.00
*     10   0.3        1500.00          150.00         445.00         707.00
         100.00     2902.00
*  !  11   0.5        1400.00          150.00         335.00         777.00
         100.00     2762.00
*     12   1.7        1400.00          150.00         335.00         777.00
         100.00     2762.00
*     13   7.0        1400.00          150.00         335.00         777.00
```

```
         100.00     2762.00
*     14   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     15   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     16   7.0       1400.00         150.00         335.00         777.00
         100.00     2762.00
*  !  17   7.0       1400.00         150.00         223.00         777.00
         100.00     2650.00
*     18   7.1       1400.00         150.00         223.00         777.00
         100.00     2650.00
*     19   7.3       1400.00         150.00         335.00         777.00
         100.00     2762.00
*     20   7.3       1400.00         150.00         215.00         848.00
         100.00     2713.00
*     21   7.3       1400.00         150.00         215.00         848.00
         100.00     2713.00
*
* Time used             : 7.48
* Nb of solutions found : 21
*
Best solution:
-----------------------------------------------------------------------------
-------------------------
DRYER                          start        end mode  mode/setup   earliness
   tardiness unperformed
-----------------------------------------------------------------------------
-------------------------
ORDER03_TEAK_DRYING_setup          0          7    0          50         0.
00        0.00          0
ORDER03_TEAK_DRYING                7         87    1         200         0.
00       49.00          0
ORDER00_PINE_DRYING_setup         79         79    0           0         0.
00        0.00          0
ORDER00_PINE_DRYING               79        160    0         100         0.
00        0.00        100
ORDER04_TEAK_DRYING_setup         87         87    0           0         0.
00        0.00          0
ORDER04_TEAK_DRYING               87        167    1         200         0.
00      329.00          0
ORDER05_TEAK_DRYING_setup        167        167    0           0         0.
00        0.00          0
ORDER05_TEAK_DRYING              167        247    1         200        33.
00        0.00          0
ORDER06_TEAK_DRYING_setup        247        247    0           0         0.
00        0.00          0
ORDER06_TEAK_DRYING              247        327    1         200         0.
00      329.00          0
ORDER07_TEAK_DRYING_setup        360        360    0           0         0.
00        0.00          0
ORDER07_TEAK_DRYING              360        480    0         100        80.
00        0.00          0
ORDER08_TEAK_DRYING_setup        480        480    0           0         0.
00        0.00          0
```

```
ORDER08_TEAK_DRYING            480        570    0         100        70.
00        0.00           0
ORDER01_PINE_DRYING_setup      570        575    0          50         0.
00        0.00           0
ORDER01_PINE_DRYING            575        645    0         100        35.
00        0.00           0
ORDER02_PINE_DRYING_setup      645        645    0           0         0.
00        0.00           0
ORDER02_PINE_DRYING            645        715    0         100         5.
00        0.00           0
ORDER09_TEAK_DRYING_setup      715        720    0          50         0.
00        0.00           0
ORDER09_TEAK_DRYING            720        810    0         100         0.
00       70.00           0

-----------------------------------------------------------------------------
-------------------------
tot_mode_cost         :      1400.00
tot_setup_cost        :       150.00
tot_earliness         :       223.00
tot_tardiness         :       777.00
tot_unperf_cost       :       100.00
total                 :      2650.00
-----------------------------------------------------------------------------
----------------------
```

Modeling a dairy plant with PPO Java API

This section models the manufacturing process of a hypothetical dairy plant, using the PPO Java™ API. First the actual manufacturing process is examined; from this the appropriate PPO model is defined and implemented in the Java API. Only some of the more interesting aspects of modeling this problem are described here, but the complete code example is located at `<PPOInstallDirectory>\examples\src\dairyplant.java`.

In this section

Overview
Lists some of the requirements to build a successful model of this plant.

The fresh dairy plant and its yogurt process
A description of the plant, process, and product.

Building the overall model structure
First, let's model some of the “big picture” aspects of the problem, such as the products, demands, stock, and other general characteristics such as the units of time and measure. The purpose of this section is to describe most of the functions necessary to create the finished products portion of the model.

Modeling the manufacturing process
With finished products code described and the overall model defined, we next focus on analyzing and modeling the actual process details of manufacturing the yogurt. The major tasks include representing the process equipment and defining the recipe boundaries.

Overview

The manufacture of yogurt at a fresh dairy plant is a complex process. To build a successful model of this process means:

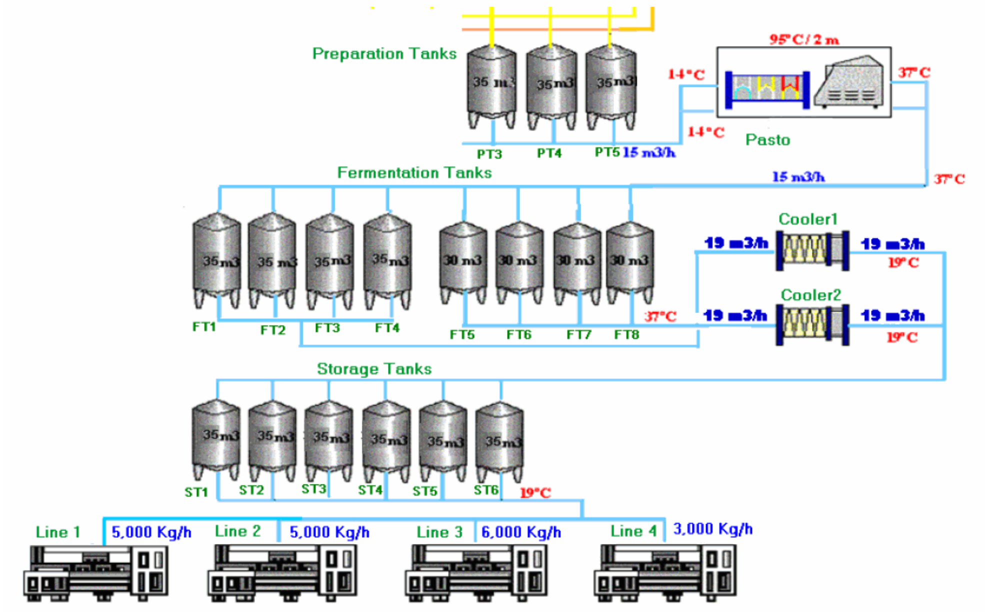
- Modeling equipment and connections,
- Determining the boundaries of the recipes,
- Deciding which materials to model (or not) and how to express the unit of measures,
- Expressing the constraints for each recipe,
- Dealing with different cleanup strategies,
- Define the setup matrix to obtain our preferred sequence,
- Deal with time zone and daylight savings time issues.

This section describes how to build such a model. The full code example itself can be found at `<PPOInstallDirectory>\examples\src\dairyplant.java`.

The fresh dairy plant and its yogurt process

First we must consider what should be the scope of the problem: The entire plant or should we create a partition between processes? Are there shared resources between production processes or not? A typical dairy plant might manufacture yogurts, drinks and cheeses. We make the assumption here that these different processes are independent and so decide to focus on the yogurt process. Otherwise, we must model the different processes together or use block planning (reserving Monday and Tuesday for yogurts, Wednesday for cheese and so forth, using calendars).

The following diagram displays the layout of the yogurt process.



In process industries, a typical plant has two different sections:

- A "process part" that creates the bulk of the semi-finished products
- A "filling/packaging" part that creates the finished products.

The diagram above represents the process part, from preparation tanks to filling lines 1 through 4. After a preparation of one hour involving milk, cream and powder in a tank, the prepared batch is pasteurized in a pasteurizer (called a *pasto*) and flows into one fermentation tank. After fermenting for 5 or 8 hours (depending on the bacteria) the fermented milk is cooled; there are two coolers, and the one used depends upon which fermentation tank was used. After cooling it is moved to a storage tank. The cold yogurt base cannot be stored for more than 24 hours before being packaged.

The yogurt base stored in the tanks is a *semi-finished intermediate* product that is fed at differing throughputs to continuous machines in the filling line. At the filling line, the flavor is added and mixed in at the last moment before packaging. (The fruit tank and mixer are not specifically represented on the diagram.) The final product is then stored on pallets in a warehouse with limited capacity.

From one semi-finished intermediate, several different finished products can be manufactured. These finished products (also called SKU for Stock Keeping Units) have different flavors and packaging. Finished products are organized into families depending on the type of semi-finished intermediate, or white mass, that they use.

Traditional yogurts need five hours of fermentation time and use two bacteria: *Lactobacillus bulgaricus* and *Streptococcus thermophilus*. The semi-finished intermediate product using only these bacteria is called “Bulgaricus White Mass”. There is also a fat-free white mass version called “Bulgaricus Light White Mass”. A third bacterium called *Bifidus* is used in making probiotic yogurts, and requires eight hours of fermentation. The corresponding intermediate is called “Bifidus White Mass.” To each of these three types of white mass, up to four types of flavor are available to create the final SKU.

So the finished products are grouped into three families:

- Those requiring Bulgaricus
- Those requiring Bulgaricus Light
- Those requiring Bifidus

And each SKU has a flavor:

- plain or natural,
- lemon,
- strawberry,
- kiwi (an allergen)

The preferred sequence within the filling and packaging lines is to fully process one family at a time, from Bulgaricus Light to Bulgaricus to Bifidus. Within each family, the preferred flavor sequence is from natural to lemon to strawberry to kiwi. Changing flavor takes 10 minutes of setup time, and changing family takes five minutes.

There are numerous cleaning requirements, all of which are handled by a cleaning in place (CIP) activity. A CIP is required when transitioning from an allergenic product to a non-allergenic product (for example, from kiwi of one family to natural flavor in the next family). There are industry safety requirements to clean the filling line at least every 24 hours. The pasto and the coolers must be cleaned every three batches. The tanks must be cleaned at least every 48 hours.

A CIP takes three hours, which includes any associated product and machine setup or changeover times. A cleaning station is used to clean the manufacturing equipment. Its role is to provide water, acid and soda, but it can clean only five pieces of equipment in parallel. Also, a “CIP line” is used to bring water, acid and soda from the station to the equipment to be cleaned. One CIP line is assigned to one piece of equipment, while other lines are shared. A CIP line cannot be used for cleaning two pieces of equipment in parallel.

The CIP lines are considered to be one of the major bottlenecks in the plant; also, changing fruit flavors takes time and cannot be neglected. However, the fruit, milk, and other raw materials and intermediates (such as pasteurized or fermented milk) are always available in sufficient quantities.

The finished products are stored in a cold warehouse near the plant. The physical capacity of this cold warehouse is 3,000 pallets.

Building the overall model structure

First, let's model some of the “big picture” aspects of the problem, such as the products, demands, stock, and other general characteristics such as the units of time and measure. The purpose of this section is to describe most of the functions necessary to create the finished products portion of the model.

In this section

Overview
Defining the model and the filling and packaging.

Setting up the model
Importing the PPO objects, instantiating the model, setting the time zone and date origin.

Defining the time buckets
Time buckets are used for planning and batching.

Defining semi-finished intermediate products
You do not need to define all intermediate products.

Defining finished products
The final consumer products are grouped by family.

Days of supply
Modeling a requirement to have sufficient stock available for demand.

Resources for finished products
The filling line and the CIP resource.

Defining the units of measure
Pallets, boxes, tons, and conversion.

Modeling a warehouse as a storage unit
Define all storable materials.

Defining the production recipes for the finished products
A recipe is necessary to create every product.

Defining cleanup recipes
Cleanups are created via recipes.

Defining initial stock
Stock elements have a size and an age.

Defining the demand
Demands should have either a revenue or nondelivery cost.

Dealing with setups and obtaining the preferred sequence
Each finished product has two main features: the semi-finished intermediate it requires and the flavor it contains

Defining the costs and the weights
Costs and weights guide the creation of the solution.

Overview

The two parts of this plant provide two different modeling challenges. In the process section, we must decide how to model different types of processes and tanks, and figure out how to connect equipment so that material flows correctly between tanks and within time constraints. In the filling and packaging section, there are numerous machine setups and changeovers to control in order to create all the possible final products; we must determine exactly how and what to manufacture. There are also cleaning requirements to model in both sections of the facility.

After first defining some overall characteristics of the model, we'll then examine the filling and packaging section, since that part drives the creation, demand, and timing of the manufacture of the final deliverable products.

Setting up the model

All PPO modeling objects can be accessed in Java™ from the `ilog.plant` package. So, first we import them:

```
import ilog.plant.*;
```

The model is instantiated using the static method `newModel`:

```
IloMSModel model = IloMSModel.newModel();
```

The `IloMSModel` instance implements a factory. All the other objects are obtained from this instance by calling `newXYZ` methods.

We must define the time granularity. We want to express times with a one-minute granularity so we define a time unit as 60 seconds:

```
model.setTimeUnit(60);
```

We also define the time zone, the date origin (or date “zero” for the problem) and the earliest start times for all activities of the model relative to this date origin:

```
import java.util.*;

String timeZoneID = “Europe/Paris”;
TimeZone timeZone = TimeZone.getTimeZone(timeZoneID);
Calendar calendar = new GregorianCalendar(timeZone);
Calendar.setTime(2006,9,28);// October 28 2006
java.util.Date origin = calendar.getTime();
model.setTimeZone(timeZoneID);
model.setDateOrigin(origin);
model.setStartMin(0); // 0 time units after the origin
```

Defining the time buckets

In order to use the planning module of PPO, we need to define time buckets. Since the plant operates in terms of stock at the end of day, we will use daily buckets. We want to do integrated planning and scheduling to cover seven days.

Note that daylight savings time occurs early on the morning of Sunday October 29th. This presents an unusual event, as 3:00 a.m. occurs twice on this particular 25-hour day. We cannot therefore declare that each day's duration is 24*60 = 1440 minutes, so we use the Java™ calendar facilities.

```
import java.text.DateFormat;

DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.FRANCE);
IloMSBucketSequence dailyBucketSequence = model.newBucketSequence();
for(int d=0; d<7; ++d) {
  Date startBucket = calendar.getTime();
  int start = model.convertDateToTime(startBucket);
  calendar.add(Calendar.DAY_OF_YEAR,1);
  Date endBucket   = calendar.getTime();
  int end = model.convertDateToTime(endBucket);
  IloMSBucket bucket = model.newBucket(start,end,dailyBucketSequence);
  bucket.setName(df.format(startBucket));
}
model.setOptimizedBucketSequence(dailyBucketSequence);
model.setDisplayedBucketSequence(dailyBucketSequence);
```

In addition to daily time buckets, it is useful to have weekly time buckets available (for example, for reviewing operational data and future planning). So we define a `weeklyBucketSequence` having a start time equal to the start of the first day and end time equal to end of the last day.

```
IloMSBucketSequence weeklyBucketSequence = model.newBucketSequence();
weeklyBucketSequence.setName("Weekly bucket sequence");
int start;
int end;
start = (dailyBucketSequence.getBucket(0)).getStartTime();
end = (dailyBucketSequence.getBucket(6)).getEndTime();
IloMSBucket weeklyBucket = model.newBucket(start, end, weeklyBucketSequence);
weeklyBucket.setName("Weekly bucket");
```

Defining semi-finished intermediate products

The finished products require (or *consume*) two main materials: flavor and semi-finished intermediates. As mentioned earlier, the fruit flavor is always available in sufficient quantities so we don't need to model it. However, the semi-finished intermediate is one of three types of processed milk, or white mass, and this material is itself the result of a manufacturing process which may have production limits. So we model the three white mass products in order to track availability. The `SetMaterialsSFP` function assigns the three intermediates to one material family (White Mass), defined with the `newMaterialFamily` method. The individual materials are created with the `newMaterial` method. Shelf life and color are set.

```
  static void SetMaterialsSFP(IloMSModel model)
  {
    //Dealing with semi-finished intermediate products
    String namesSFP[] = { "Bifidus White Mass", "Bulgaricus White Mass",
"Bulgaricus Light White Mass" };
    String identifiersSFP[] = { "bifidus", "bulgaricus", "bulgaricusLight" };

    int shelfLifeSFP[] = { 30 * 24 * 60, 30 * 24 * 60, 30 * 24 * 60 }; // 30
days of shelf life
    String colorsSFP[] = { "wheat", "lightgreen", "paleturquoise" };

    //Families of products and colors
    IloMSMaterialFamily familySFIProduct = model.newMaterialFamily();
    familySFIProduct.setName("White Mass");
    familySFIProduct.setType("product type");

    for (int i = 0; i < identifiersSFP.length; i++)
    {
      IloMSMaterial mat = model.newMaterial();
      mat.setIdentifier(identifiersSFP[i]);
      mat.setName(namesSFP[i]);
      mat.setShelfLife(shelfLifeSFP[i]);
      mat.setColor(colorsSFP[i]);
      familySFIProduct.add(mat);
    }
  }
```

The rationale for defining only these three semi-finished white mass products, and not all intermediates such as prepared or fermented milk, is discussed further in *Modeling the intermediate materials and process activities.*

Defining finished products

The final products are defined in the function `SetMaterialsFP` which is similar to the function for the semi-finished products. However since each member of the white mass family has four flavors, we end up with 12 final products. For easier sorting in the GUI we group the final products into three families corresponding to white mass.

```
String familyNamesFP[] = { "Bifidus Finished Products", "Bulgaricus Finished
Products", ... };
String namesFP[][] = { {"Bifidus lemon", "Bifidus plain", "Bifidus strawberry",
 "Bifidus kiwi" },
                               {"Bulgaricus lemon", ... },
                               { "bulgaricusLightLemon", ... }}
```

This function also sets `identifiersFP`, used elsewhere in the code to identify the finished product.

The finished products have a maturity period to ensure that products are not shipped early, and like the intermediates, a shelf life period. The colors are set by flavor, not family type. Available colors are the X11 color set, in lowercase with no white space. A complete list of X11 colors is available from various internet sites.

```
int maturityFP[] = { 24 * 60, 24 * 60, 24 * 60 }; // one day of hard quarantine
int shelfLifeFP[] = { 30 * 24 * 60, 30 * 24 * 60, 30 * 24 * 60 }; // 30 days
of shelf life
String colorsFP[] = { "yellow", "antiquewhite", "purple", "green" };
```

These characteristics are set to each material in a loop as done for the semi-finished products, although here two `for` loops are necessary to populate the material family and the four materials within each family.

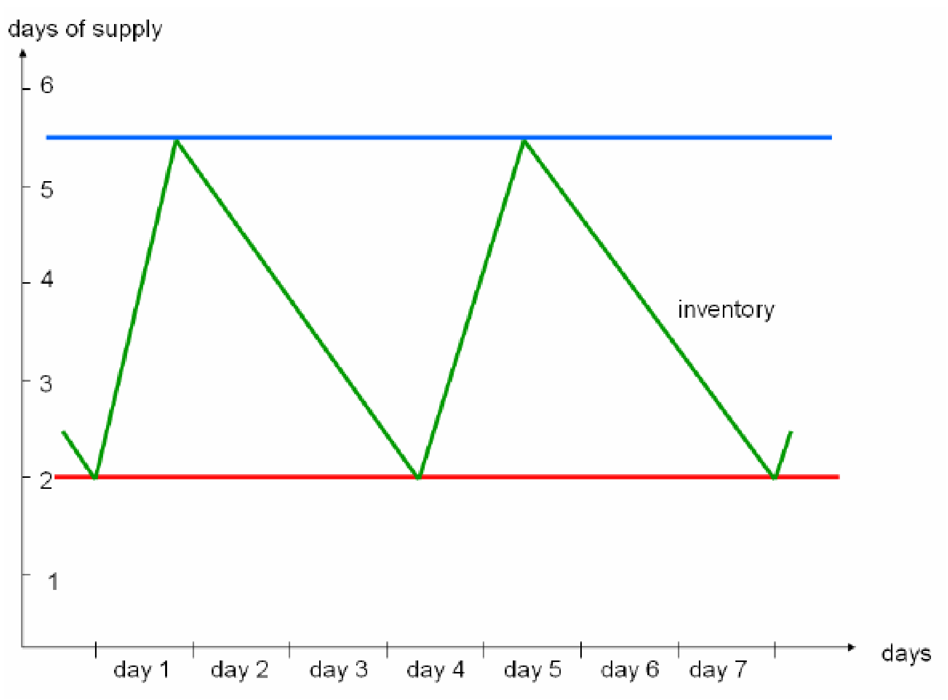
Additionally with the finished products we want to make sure we always have inventory on hand to meed demand; so we need to define a days of supply target minimum and maximum. This stock coverage is further discussed in the next section *Days of supply*. Following is the code excerpt from the material `for` loop; the maturity is pulled from the array created previously, and the days of supply for all products is between two to five and a half days.

```
        mat.setMaturity(maturityFP[i]); // one day of hard quarantine
        ...
        //Days of supply
        mat.setDaysOfSupplyTargetMin(2.0);
        mat.setDaysOfSupplyTargetMax(5.5);
```

The rest of the function `SetMaterialsFP` is similar to the function described in *Defining semi-finished intermediate products*. Please see the code example for full details.

Days of supply

The supply chain imposes the requirement to have two days supply of each final product. That is, every night the stock of products must be sufficient to cover two days of demand. We also need two different expiration dates per week; in other words, two productions per week. There is a simple way to define the minimum and maximum stock in PPO using the minimum and maximum days of supply for each SKU. In this case the minimum is 2 and the maximum is 2 + 7/2 = 5.5. The ideal inventory level with two productions a week follows this figure.



We coded these targets in the previous section*Defining finished products*. Note that these bounds on the inventory are soft constraints. We will see later how to penalize solutions when the inventory goes below or above these bands. PPO allows creating arbitrarily complex piecewise-linear functions of quantity for inventory excess and deficit. Using days of supply is a short-cut. Also, you can create a stock corridor based on industry-standard service level types, and even consider variability in demand. This is not demonstrated here, but see *Using service levels, lead time, and demand variability to manage stock levels* for more information.

Resources for finished products

The filling lines are from the function `SetResourcesFP`. This function sets names and identifiers of course, but also sets capacity, rank (used to place the resource in the **Gantt Diagram** of the GUI) and the category (used in the **Plant Layout** view). Also the filling lines are grouped into one super resource to lighten the planning problem. Here's the entire function:

```
  static void SetResourcesFP(IloMSModel model)
  {
    String namesResFP[] = { "Line 1", "Line 2", "Line 3", "Line 4" };
    String identifiersResFP[] = { "line1", "line2", "line3", "line4" };

    IloMSResource lines = model.newResource(identifiersResFP.length);
    lines.setIdentifier("lines");
    lines.setName("Lines");
    lines.setRank(1000);

    for (int i = 0; i < identifiersResFP.length; i++)
    {
      IloMSResource res = model.newResource(1);
      res.setRank(1000 + i + 1);
      res.setIdentifier(identifiersResFP[i]);
      res.setName(namesResFP[i]);
      res.setSuperResource(lines);
      res.setCategory("packer"); // For display in Plant layout view with
appropriate symbol
    }
  }
```

In the function `SetResourcesCleaning`, the CIP station is represented by a resource of capacity five. The CIP lines are grouped into a super resource to lighten the planning problem.

```
IloMSResource cipStation = model.newResource(5);
...

IloMSResource cipLines = model.newResource(6);
...
    for (int i = 0; i < 6; i++)
    {
      IloMSResource cipLine = model.newResource(1);
      cipLine.setIdentifier("cipLine" + (i + 1));
      cipLine.setName("CIP Line " + (i + 1));
      //Grouping resources
      cipLine.setSuperResource(cipLines);
    }
```

Many more resources are required for the intermediate products; this is covered in later sections once decisions are made regarding how much of the manufacturing process to model.

Defining the units of measure

The production of one pallet of Bulgaricus Lemon requires one-half metric ton, or 500 kg, of Bulgaricus White Mass. Depending on the filling line, the throughput ranges from 3,000 to 6,000 kg/h.

To prevent numerical problems we avoid using very different units for production and consumption. We will use the pallet as the primary unit for the finished products, and the ton as primary unit for white masses. To define the units of measure we use the `IloMSUnit` class. Two types of units of measure must be distinguished, and we do this in the `SetUnitsSFP` function. One unit has a dimension that can be converted into the metric system:

```
IloMSUnit ton = model.newUnit();
ton.setName("ton");
ton.setDimension(IloMSDimension.DimensionMass);
ton.setStandardConversion(0.001); // conversion from standard unit (kg)
```

The other type converts depending upon the material:

```
IloMSUnit piece = model.newUnit();
piece.setName("piece");
IloMSUnit box = model.newUnit();
box.setName("box");
box.setCheckingTolerance(1.0); // the default is 0.01
IloMSUnit pallet = model.newUnit();
pallet.setName("pallet");
```

To filter out insignificant violations from appearing in the GUI violation panel, we use the `setCheckingTolerance` API on the display unit. It has no influence on computation. Here it was used to have a checker warning about a mismatch in quantity only when it is greater than one box.

For each material we must define the primary unit (in which all computation takes place) and any secondary units used for display in the GUI. By default, this is the primary unit but we can instead use a secondary unit if a conversion to the primary is provided. In this example a pallet of final product is composed of 48 boxes, there are 54 pieces per box, and so 2592 pieces per pallet.

```
  for (int i = 0; i < identifiersFP.length; i++)
    {
      IloMSMaterial mat = model.getMaterialByIdentifier(identifiersFP[i]);
      mat.setPrimaryUnit(pallet);
      mat.addSecondaryUnit(box, 48, 1); //48 boxes/pallet
     mat.addSecondaryUnit(piece, 2592, 1); //54 pieces/box=2592 pieces/pallet

      mat.setDisplayUnit(pallet);
    }
```

Modeling a warehouse as a storage unit

The finished products are stored in a cold warehouse near the plant. The physical capacity of this cold warehouse is 3,000 pallets.

We create a storage unit able to store all the finished products in the `SetStorageUnitsFP` function.

```
IloMSStorageUnit warehouse = model.newStorageUnit();
...
warehouse.setQuantityMax(3000);
```

Within a loop the materials are added as storable materials in the `warehouse`.

```
      IloMSMaterial mat = model.getMaterialByIdentifier(identifiersFP[i]);
      warehouse.addStorableMaterial(mat);
```

Defining the production recipes for the finished products

A recipe is a mold or template of production orders. It defines a canonical production order that is multiplied by the batch size when instantiating the production order. In this section we'll deal with the finished products alone; the intermediate recipes are described in several sections of *Modeling the manufacturing process*.

Production of final products is a unique activity that continuously consumes semi-finished intermediate and continuously produces the finished product. The speed depends on the filling line used. First, in the `SetRecipesFP` function we create a recipe for every final product with minimal and maximal batch sizes.

```
    String recipesIdentifiersFP[] = { "BifidusLRecipe", "BifidusPRecipe", ...
.. , "BulgaricusLightKRecipe"};
    String recipesNamesFP[] = { "Bifidus lemon", "Bifidus plain", .... ,
"Bulgaricus Light kiwi" };
    for (int i = 0; i < recipesIdentifiersFP.length; i++)
    {
      IloMSRecipe recipe = model.newRecipe();
      recipe.setIdentifier(recipesIdentifiersFP[i]);
      recipe.setName(recipesNamesFP[i]);
     recipe.setBatchSizeMin(10); // To avoid producing fewer than 10 pallets/
production order
      recipe.setBatchSizeMax(50); // Maximal production of a production order

    }
```

This is followed later by the function `SetProductionsFP` where we express the fact that the activity continuously produces pallets of product that are directly stored in the warehouse. A production order of batch size one produces one pallet.

```

    IloMSMaterialProduction production = model.newMaterialProduction(mode,
material, 1);
    production.setName("Production of " + material.getName() + " on " + res);

    production.setContinuous(true);
    production.setStorageUnit(warehouse);
```

Note that if the recipe were producing multiple products (or ingredients) we would have to define the primary product (or primary ingredient) for GUI purposes.

We must specify the setup states required by this activity. The particular setup requirements are detailed later in *Dealing with setups and obtaining the preferred sequence*, but for now assuming they are known, the function `SetActivitiesFP` codes the setups, colors, and activity names. The string `semiTypes` identifies the white mass, and the activities are populated within a double loop to account for both setup features. The `setName` method ensures that the activity label displayed in Gantt will contain product name and quantity.

```

        activity.setColor(mat.getColor());
        activity.setSetupState("semi-type", semiTypes[i]);
        activity.setSetupState("flavor", flavors[j]);
        activity.setName("^p ^q");
```

The function `SetModesFP` defines the modes for each activity; the filling line used and its processing time and cost. The assignment preference is realized using the processing cost; each alternative mode has a different processing cost. If we prefer to produce KPI Lemon on line 3 we get a smaller processing cost. The index is `[materialSFP][flavorFP][line]`. The processing time is based on 5000 kg per hour, or 6 minutes for 500 kg, based on one-half ton of white mass for one pallet of KPI.

```
static void SetModesFP(IloMSModel model)
{
  String matSFP[] = { "Bifidus", "Bulgaricus", "BulgaricusLight" };
  String flavorFP[] = { "L", "P", "S", "K" };
  String resIdFP[] = { "line1", "line2", "line3", "line4" };
  double variableProcTimeFP[] = { 6, 6, 5, 10 };//5000kg/h -> 6min for 500kg
(0.5 ton of white mass for 1 pallet of KPI)
  double variableProcCostFP[][][] = {{{5,5,1,5},{5,5,5,5},{5,5,5,5},{5,5,5,5}
}, // Bifidus
                                     {{5,5,1,5},{5,5,5,5},{5,5,5,5},{5,5,5,5}
}, // Bulgaricus
                                     {{5,5,1,5},{5,5,5,5},{5,5,5,5},{5,5,5,5}
}}; // Bulgaricus Light

  for (int m = 0; m < matSFP.length; m++)
  {
    for (int f = 0; f < flavorFP.length; f++)
    {
      for (int r = 0; r < resIdFP.length; r++)
      {
        IloMSMode mode = model.newMode((IloMSActivity)model.
getActivityPrototypeByIdentifier("act" + matSFP[m] + flavorFP[f]));
        mode.setName(matSFP[m] + flavorFP[f] + " on " + resIdFP[r]);
       model.newResourceConstraint(mode, model.getResourceByIdentifier(resIdFP
[r]), 1, true);
        mode.setVariableProcessingTime(variableProcTimeFP[r]);
        mode.setVariableCost(variableProcCostFP[m][f][r]);
      }
    }
  }
}
```

Defining cleanup recipes

Cleaning in PPO is realized using production orders of a cleanup recipe. A cleanup recipe is composed of an activity with a cleaning status `IloMSCleaningStatus.Cleaning`. The cleaning time is the processing time of this activity. The cleaning cost is the processing cost of the cleaning recipe. It must dominate any cost in the setup matrix.

First the cleanup recipe for cleaning lines is created in the function `SetRecipesCleaning`.

```
IloMSRecipe cleanupRecipeFillingLines = model.newRecipe();
```

Then the cleaning activity prototype is defined. Note that a cleaning activity does not require a product or flavor, but we use dummy states for the setup features.

```
IloMSActivity cleaningActivity = model.newActivityPrototype
(cleanupRecipeFillingLines);
cleaningActivity.setIdentifier("cleaningActivityFillingLines");
cleaningActivity.setName("CIP");
cleaningActivity.setSetupState("semi-type", "dummySemi");
cleaningActivity.setSetupState("flavor", "dummyFlavor");
cleaningActivity.setCleaningStatus(IloMSCleaningStatus.Cleaning);
cleaningActivity.setColor("gray");
```

On the Gantt Diagram view in the GUI, cleaning activity objects appear overlaid with a bubble pattern. Since in this example the cleaning processing time hides the setup time, and the setup activities are typically represented in gray, we also define cleaning activities as gray.

Then the function `SetModesCleaning` takes care of the rest: Setting the maximum time before a cleanup, applying the cleanup recipe to the resource, linking each line to a CIP line, adding the resource constraints, and setting the cost and time for the operation.

```
    String identifiersLines[] = { "line1", "line2", "line3", "line4" };
    int maxTimeBeforeCleanupLine = 24 * 60; // Lines must be cleaned at least
 every 24 hours (24*60)
    for (int r = 0; r < identifiersLines.length; r++)
    {
      IloMSResource res = model.getResourceByIdentifier(identifiersLines[r]);

      res.setCleanupRecipe(cleanupRecipeFillingLines);
      res.setMaxTimeBeforeCleanup(maxTimeBeforeCleanupLine);
      //Define modes
      IloMSMode mode = model.newMode(cleaningActivityFillingLines);
      // filling line  1 is linked to cip line 1 etc.
      int cipLineIndex = r % identifiersCipRes.length;
      mode.setName("cleaning " + res.getName() + " with " + identifiersCipRes
[cipLineIndex]);
      model.newResourceConstraint(mode, res, 1, true);
      model.newResourceConstraint(mode, cipStation, 1, false);
      model.newResourceConstraint(mode, model.getResourceByIdentifier
(identifiersCipRes[cipLineIndex]), 1, false);
```

```
      mode.setVariableCost(10000);
      mode.setFixedProcessingTime(180);
    }
  }
```

We use a variable cost in this cleanup recipe since it covers the cleaning of different equipment in the plant, and so the cleanups are penalized proportionally to the time it requires. Note that cleaning also requires the CIP Station as a resource.

Cleanups are also required for certain product transitions, and that's covered with the setup matrix defined in *Dealing with setups and obtaining the preferred sequence*.

The cleanup code for processing equipment is similar and covered later in *Cleaning policy*.

Defining initial stock

Use the `IloMSProcurement` class to create stock elements with an age.

```
// 40 pallets of Bifidus Lemon produced October 27 12:00
// Each stock element is modeled as a procurement received in the past
IloMSProcurement stock = model.newProcurement(bifidusLemon,40);
stock.setProductionTime(-770);
stock.setReceiptTime(-770);
stock.setStorageUnit(warehouse);
model.newProcurementToStorageArc(bifidusLemon, stock.getQuantity(), stock);
```

Defining the demand

Demand is expressed with instances of `IloMSDemand`. We can express the fact that the forecast for Bifidus Lemon on Oct 31 is to be delivered at the earliest on Oct 31 00:00 and at the latest on Nov 2 00:00. We penalize a late delivery on Nov 1 using a due date and a tardiness variable cost.

```
// 110 pallets for Bifidus Lemon are due for Nov 1 0:00
IloMSDemand demand = model.newDemand(bifidusLemon,110);
demand.setName("Forecast Bifidus Lemon Nov 1");
demand.setDeliveryStartMin(model.getBucket(3).getStartTime());
demand.setDeliveryEndMax(model.getBucket(5).getStartTime());
IloMSDueDate dueDate =
 model.newDueDate(demand, model.getBucket(4).getStartTime());
```

The due date object carries the tardiness variable cost incurred for each time unit the delivery is late.

```
dueDate.setTardinessVariableCost(1.0);
```

We create a nondelivery variable cost, a cost for failing to deliver product. It is a function of the unsatisfied quantity as expressed in the primary unit of the material. We must define the storage unit for the demand as well.

```
demand.setNonDeliveryVariableCost(10000);
demand.setStorageUnit(warehouse);
```

Either revenue or nondelivery costs must be provided to create an incentive to meet a demand; otherwise, the optimal solution may consist of delivering to stock or producing nothing at all, rather than fulfilling demand. More details about setting cost values follows in upcoming sections.

Dealing with setups and obtaining the preferred sequence

Each finished product has two main features: the semi-finished intermediate it requires and the flavor it contains. As mentioned earlier, there is a preferred sequence to process one intermediate family at a time (Bulgaricus Light to Bulgaricus to Bifidus) and then one flavor at a time (from natural to lemon to strawberry to kiwi). We will use several setup matrices to model this preferred sequence.

We want to create long campaigns of several products requiring the same semi-finished intermediate. Then inside a campaign, we process flavors based on increasing darkness of color and finishing with the allergen products.

PPO allows you to define two setup features, so that you can avoid creating a big matrix defining the Cartesian product of all possibilities. We create a matrix for the setup times for the two setup features introduced earlier, semi-type and flavor.

Semi-type setup times	Bulgaricus	Bulgaricus Light	Bifidus
Bulgaricus	0	5	5
Bulgaricus Light	5	0	5
Bifidus	5	5	0

Flavor setup times	plain	lemon	strawberry	kiwi
plain	0	10	10	10
lemon	10	0	10	10
strawberry	10	10	0	10
kiwi	10	10	10	0

For both setup features, the need for cleanup after an allergen must also be specified.

Semi-type cleanup	Bulgaricus	Bulgaricus Light	Bifidus
Bulgaricus	FALSE	FALSE	FALSE
Bulgaricus Light	FALSE	FALSE	FALSE
Bifidus	FALSE	FALSE	FALSE

Flavor cleanup	plain	lemon	strawberry	kiwi
plain	FALSE	FALSE	FALSE	FALSE
lemon	FALSE	FALSE	FALSE	FALSE
strawberry	FALSE	FALSE	FALSE	FALSE
kiwi	TRUE	TRUE	TRUE	FALSE

We define setup costs to direct optimization towards our preferred manufacturing sequence. First let’s define the matrix for flavors. As a rule of thumb we increment by 100 on each row and decrement by 100 on each column, except for the diagonal. The largest value is the sum of the first row plus 100.

Flavor setup costs	plain	lemon	strawberry	kiwi
plain	0	100	200	300
lemon	700	0	100	200
strawberry	600	700	0	100
kiwi	500	600	700	0

Next we define setup costs for the semi-finished intermediate families. These costs must dominate the flavor costs; using the worst sequence in the flavors should not override our preferred sequence with the family products. So we apply the same rule as before, starting with a bigger cost (2200) than the cost for the worst sequence (3*700=2100) from the flavor matrix. Remember that the setup features are additive.

Semi-type setup costs	Bulgaricus	Bulgaricus Light	Bifidus
Bulgaricus	0	2200	2300
Bulgaricus Light	4600	0	2200
Bifidus	4500	4600	0

With the resulting code from the function `SetSetups`:

```
IloMSSetupMatrix semiTypeMatrix = model.newSetupMatrix();
semiTypeMatrix.setSetup("bulgaricus", "bulgaricus", 0, 0, false);
semiTypeMatrix.setSetup("bulgaricus", "bulgaricus-light", 5, 2200,false) ;
…
IloMSSetupMatrix flavorMatrix = model.newSetupMatrix();
```

```
flavorMatrix.setSetup("plain", "plain", 0, 0, false);
flavorMatrix.setSetup("plain", "lemon", 10, 100, false);
```

Then the setup matrix and initial state used by the line resource (`res`) for both semi-finished product and flavor is specified in a loop.

```

    res.setSetupMatrix("semi-type", semiTypeMatrix);
    res.setSetupMatrix("flavor", flavorMatrix);
    res.setInitialSetupState("semi-type", InitialSetupStateSemiType[r]);
    res.setInitialSetupState("flavor", InitialSetupStateFlavor[r]);
```

The scheduling engine uses these exact costs, times, and cleanup details on the sequence it finds. The planning engine, on the other hand, uses different levels of approximations of the setups. You can define for each resource and for a time extent a different level of approximation among:

```

  IloMSPlanningSetupModel.PlanningSetupModelNoSetup
  IloMSPlanningSetupModel.PlanningSetupModelPerBucketPerRecipe
  IloMSPlanningSetupModel.PlanningSetupModelPerBucketPerSetupFeature
  IloMSPlanningSetupModel.PlanningSetupModelCrossBucketPerRecipe
  IloMSPlanningSetupModel.PlanningSetupModelCrossBucketPerSetupFeature
```

The more precision used, the higher the cost in algorithmic complexity. In this example we don’t need to be very precise, so we choose the “per bucket per recipe” approximation:

```

res.setPlanningSetupModel(
  IloMSPlanningSetupModel.PlanningSetupModelPerBucketPerRecipe,
  model.getBucket(0).getStartTime(),
  model.getBucket(model.getNumberOfBuckets()-1).getEndTime());
```

Definitions of these approximations are listed with the `PPO_RESOURCE_SETUP_MODEL` table in the *PPO Data Schema*.

Defining the costs and the weights

We want to define costs and weights that balance these objectives:

- ♦ Fulfill the demand,
- ♦ Respect the minimum and maximum days of supply in the inventory,
- ♦ Minimize setup costs including cleaning,
- ♦ Respect the resource assignment preference.

The largest setup cost is the cleaning cost which is 10,000. Let’s examine a production order of Bifidus Lemon at its minimum batch size (10 pallets) and one day of production (240 pallets).

If we take 1000 as a target variable cost, the inventory and inventory deficit cost for 10 pallets for one day below the minimal inventory will be 10,000. For one day’s production of 240 pallets, it will be 24,000. So the inventory and inventory deficit costs will be greater than the setup cost.

We must then use at least 10,000 for the unsatisfied demand cost. Operating 10 days below the minimal inventory is 240,000, which is dominated by 2,400,000 so it is more appropriate to ship rather than to keep in stock.

Process cost is a secondary objective; it is multiplied by the processing time so let’s use from 60 to 1440 for the smallest to largest production orders. We can take between 1 and 10 for encoding the different preferences.

The tardiness cost for one day of lateness for one pallet is 1440.

Cost type	10 pallets equivalent	240 pallets (one day) equivalent
Nondelivery	100,000	2,400,000
Tardiness	14,400	345,600
Inventory deficit	10,000	240,000
Inventory	10,000	240,000
Setup	10,000	10,000
Processing	600	14,400

We penalize the situation where the inventory goes above or below the minimum and maximum days of supply for one pallet of Bifidus Lemon:

```
bifidusLemon.setTargetMinVariableCost(100.0);
bifidusLemon.setTargetMaxVariableCost(100.0);
```

Each pallet left unsatisfied by the plan is then penalized by 10,000:

```
demand.setNonDeliveryVariableCost(10000);
```

Each pallet exiting the inventory late by X time units is also penalized by X:

```
dueDate.setTardinessVariableCost(1.0);
```

We must put a global weight for each component of the costs. From the `SetOptimizationProfile` function:

```
IloMSOptimizationProfile profile = model.newOptimizationProfile();
profile.setName(“ALL PRODUCTS”);
model.setCurrentOptimizationProfile(profile);

profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalNonDeliveryC
ost”),1.0);
profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalTardinessCos
t”),1.0);
profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalInventoryCos
t”),1.0);
profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalInventoryDef
icitCost”),1.0);
profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalSetupCost”)
,
1.0);
profile.setWeight(model.getOptimizationCriterionByIdentifier(“TotalProcessingCo
st”),1.0);

profile.setPlanningRequired(true);
profile.setBatchingRequired(true);
profile.setSchedulingRequired(true);
```

We've now touched upon most of the code that would be required to launch an optimization involving the finished products alone. Assuming a completed model you could then launch a solve:

```
model.solve();
```

An easy way to create a .csv file containing the problem (and the solution if any) is to call:

```
model.write(“finishedProducts.csv”,true);
```

You can then open the .csv file from the GUI and solve the problem there. However, in this example, we continue as we want to include the semi-finished products.

Modeling the manufacturing process

With finished products code described and the overall model defined, we next focus on analyzing and modeling the actual process details of manufacturing the yogurt. The major tasks include representing the process equipment and defining the recipe boundaries.

In this section

Representing process equipment
There are continuous machines and tanks in the process area of the plant.

How to represent equipment connections
Essentially, how to represent the pipes and conduits in the plant.

Modeling the intermediate materials and process activities
Recipe granularity depends upon batch size and possible mergers.

Internal constraints of the recipe
Several activity constraints are described.

Tanks as storage units
Tanks are used in various ways; for preparation, fermentation, cooling, and storage.

Production, consumption and destination
Expressing consumption of an intermediate.

Cleaning policy
Code to create the cleaning activities.

Calendars and breaks
Calendars are used to model temporal changes in resource availability and productivity.

Representing process equipment

In the process part, we typically find two types of equipment: continuous machines and buffers generally called “tanks”. The process can be represented as a sequence of tanks and continuous machines. The continuous machines can be seen as pumps forcing the speed of emptying the tank of the previous step and forcing the speed of filling the tank of next step.

In PPO, equipment used by production orders is typically represented by an instance of the `IloMSResource` class. As a resource may be able to execute one or several operations at a time, we must define the *capacity* of the resource. If a resource can handle only one action or item at a time it is called *unary* and its capacity is one. Note that this notion of capacity has nothing to do with the notion of volume of tanks. The capacity concept we use here is to be understood as “degree of possible parallelism”.

This example includes a pasteurizer (pasto), cooler, and preparation, storage, and fermentation tanks. The fermentation tanks are of two different capacities. The function `SetResourcesSFP` is used to model all of these resources. Here's the code for modeling preparation tanks; each has a capacity of one but combined as a super resource have a capacity of three.

```
    IloMSResource preparationTanks = model.newResource(3);
    preparationTanks.setIdentifier("preparationTanks");
    preparationTanks.setName("Preparation Tanks");
    preparationTanks.setRank(1);
    for (int i = 0; i < 3; i++)
    {
      IloMSResource preparationTank = model.newResource(1);
      preparationTank.setIdentifier("preparationTank" + (i + 1));
      preparationTank.setName("Preparation Tank " + (i + 1));
      //Grouping resources as super resources to simplify planning and master
data
      preparationTank.setSuperResource(preparationTanks);
     preparationTank.setCategory("tank"); // For display in Plant layout view
with appropriate symbol
    }
```

Similar code applies to modeling the pasteurizer, fermentation tanks, coolers, and storage tanks. Each code segment starts with the `newResource` method on either the singular resource or the corresponding super resource:

```
    IloMSResource pasto = model.newResource(1);
    ...
    IloMSResource fermentationTanks30t = model.newResource(4);
    ...
    IloMSResource fermentationTanks35t = model.newResource(4);
    ...
    IloMSResource cooler = model.newResource(1);
    ...
    IloMSResource storageTanks = model.newResource(6);
```

You may recall that there are two coolers in this example, but they are not combined into a super resource because they do not share the same connectivity; so each is defined separately with its unary capacity. The two sizes of fermentation tanks are grouped into two different super resources, using the index of the loop:

```
     if (i < 5)
        fermentationTank.setSuperResource(fermentationTanks35t);
      else
        fermentationTank.setSuperResource(fermentationTanks30t);
```

See the example for the entire function.

Note: Cleaning and setup activities require the same primary resource as the corresponding production activity.

These resources must be now be connected.

How to represent equipment connections

Not all paths between resources are possible in the factory. If a batch is fermented in the larger fermentation tank, it can only be cooled in cooler number one, and the smaller fermenter can only use cooler number two. Also, the storage tanks of the manufacturing facility must be connected to the filling lines of the packaging area.

```
    preparationTanks.addConnectedResource(pasto);
    pasto.addConnectedResource(fermentationTanks35t);
    pasto.addConnectedResource(fermentationTanks30t);
    IloMSResource cooler1 = model.getResourceByIdentifier("cooler1");
    IloMSResource cooler2 = model.getResourceByIdentifier("cooler2");
    fermentationTanks35t.addConnectedResource(cooler1);
    fermentationTanks30t.addConnectedResource(cooler2);
    cooler1.addConnectedResource(storageTanks);
    cooler2.addConnectedResource(storageTanks);
    // To connect process part storage tank and filling/packaging line part:
    String resLines[] = { "line1", "line2", "line3", "line4" };
    for (int r = 0; r < resLines.length; r++)
     storageTanks.addConnectedResource(model.getResourceByIdentifier(resLines
[r]));
```

Note that if all combinations are possible it is pointless to define connections.

Modeling the intermediate materials and process activities

Next we model the production processes. In PPO the recipe is a template of production orders consuming and producing the same materials. We have several choices in the level of granularity, because it is possible to group together several steps into a single recipe to minimize the number of production orders to manage.

To know the best level of granularity, consider the cardinality constraints between batches at each step. A batch is prepared in a preparation tank then flows into one batch of fermentation through the pasteurizer, and then into one batch in one of the many storage tanks through one cooler. There is no split or merge of batches from the preparation tank through to the storage tank. Then the white mass is split into many orders of finished products. So, in this yogurt process, we have a 1:1 relation on the whole process part, and then a 1:n relationship at the boundary between process and filling/packaging.

So we can create recipes of semi-finished white mass that include all the process activities from preparation to storage. This means that we don't have to model all the intermediate products: the prepared milk, the fermented milk, and so forth. We model only the "final" semi-finished materials: the Bulgaricus White Mass, the Bularicus Light White Mass and the Bifidus White Mass. The semi-finished products were modeled with this approach in *Defining semi-finished intermediate products*.

The recipe for white mass contains five steps:

1. Preparation
2. Pasteurization
3. Fermentation
4. Cooling
5. Storage

Note that at the fermentation step there are clearly two groups of tanks of different volume capacity. This will force us to have two different recipes per semi-finished product, one using the 30 ton fermentation tanks and one using the 35 ton fermentation tanks. The batch size min is 20 tons (to ensure the helix soaks in the storage tank).

We will use as recipe size the same unit of measure (ton) as used for the semi-finished product; also we consider that one liter equals 1kg.

Each step alternatively uses tanks and continuous machines. Continuous steps such as pasteurization and cooling are unique activities with a processing time purely proportional to the batch size.

There are three steps to using a tank: Filling it; the operation or activity itself (such as fermentation); and the emptying of the tank. We can simplify this to two steps by using a variable processing time to fill the tank (depending on the batch size) and a fixed processing time for the activity (such as fermentation). Likewise we simplify the storage step with a filling activity and a storage activity that ends when the tank is empty.

Taking the preparation tank as a specific example, we have an activity called `prepFill` which takes one hour that includes filling. There is also `prepEmpty` which has a processing time dependent upon the next operation which is pasteurization. At a minimum, we assign one minute to this task. The maximum for one full batch is to empty at the pasto speed plus a maximum of two hours waiting before emptying.

Finally, if several alternative resources are available for an activity, we need as many modes as there are resources for this activity. However, when resources are grouped into a super resource, we create only one mode per super resource at the level of the recipe activities. PPO will automatically generate modes for subresources at the production order level.

So looking at the preparation tank code from the `SetModesSFP` function:

```
  static void SetModesSFP(IloMSModel model)
  {
    String matSFP[] = { "Bif", "Bulg", "BulgLight" };

    double batchSizeMin = 20.0; // to ensure the helix soaks in the tanks
    double batchSizeMax = 35.0; // maximum capacity of preparation and storage
 tanks
    double batchSizeMaxFermentation[] = { 30.0, 35.0 }; // When using
fermentation tank 35 or 30 tons

    String actNamesPrep = "Preparation";
    String resIdPrep = "preparationTanks"; // resources for preparation
    String actIdPrep[] = { "prepFill", "prepEmpty" }; // activities of
preparation
    int pastoTimeMax = 1 + (int)(60 * 35 / 15.);
    int fixedProcTimeMinPrep[] = { 60, 1 };
    int fixedProcTimeMaxPrep[] = { 60, pastoTimeMax + 120 };

    for (int a = 0; a < actIdPrep.length; a++)
    { // one mode per activity
      for (int m = 0; m < matSFP.length; m++)
      { // one mode per material to produce
        IloMSResource res = model.getResourceByIdentifier(resIdPrep);
        IloMSMode mode = model.newMode((IloMSActivity)model.
getActivityPrototypeByIdentifier(actIdPrep[a] + matSFP[m]));
        model.newResourceConstraint(mode, res, 1, true);
        mode.setName(actNamesPrep + " " + matSFP[m] + " on " + res.getName())
;
        mode.setFixedProcessingTimeMin(fixedProcTimeMinPrep[a]);
        mode.setFixedProcessingTimeMax(fixedProcTimeMaxPrep[a]);
        mode.setBatchSizeMin(batchSizeMin);
        mode.setBatchSizeMax(batchSizeMax); // 35t preparation tanks.
      }
    }
```

Please see the full code example for the code required to model the other process activities.

Internal constraints of the recipe

The process manufacturing activities have three types of internal constraints:

- activity compatibility constraints
- precedence constraints
- spanning constraints

We must state that certain activities must be processed on connected resources.

```
model.newActivityCompatibilityConstraint(prepEmpty, pasteur,
IloMSActivityCompatibilityType.ConnectedPrimaryResources);
model.newActivityCompatibilityConstraint(pasteur, fermFill,
IloMSActivityCompatibilityType.ConnectedPrimaryResources);
model.newActivityCompatibilityConstraint(fermEmpty, cooling,
IloMSActivityCompatibilityType.ConnectedPrimaryResources);
model.newActivityCompatibilityConstraint(cooling, storFill,
IloMSActivityCompatibilityType.ConnectedPrimaryResources);
```

We also need to define an activity chain to prevent interruption of the process. The activities of an activity chain must be performed on the same resource in a defined order, and no other activity not belonging to the chain can be in between.

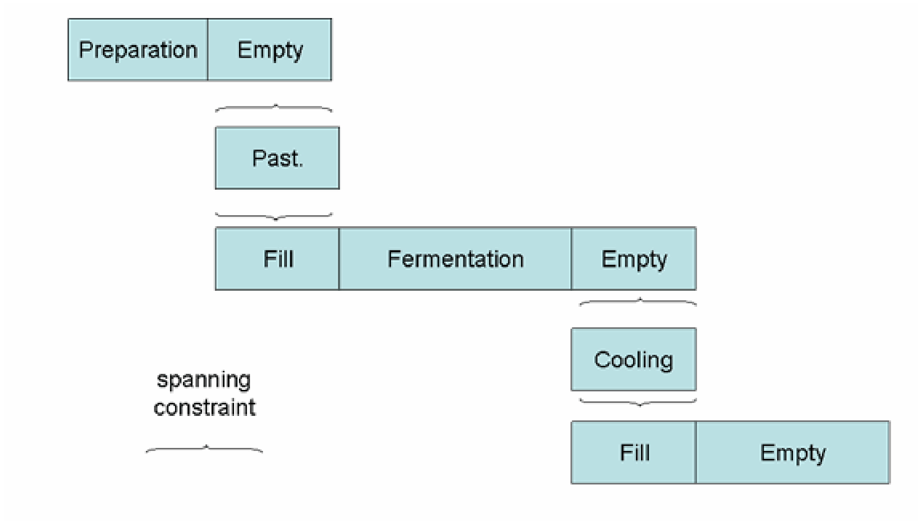
```
IloMSActivityChain fermentation = model.newActivityChainPrototype(recipe);
fermentation.appendActivity(fermFill);
fermentation.appendActivity(fermOp);
fermentation.appendActivity(fermEmpty);
```

Also certain activities must occur one right after the previous with zero delay:

```
model.newPrecedence(fermFill, fermOp, IloMSPrecedenceType.EndToStart, 0, 0);
model.newPrecedence(fermOp, fermEmpty, IloMSPrecedenceType.EndToStart, 0, 0);
```

We must then synchronize the emptying (or filling) of a step with the speed of the continuous machine used at the next (or previous) step; this is the spanning constraint.

```
prepEmpty.addSpannedActivity(pasteur);
fermFill.addSpannedActivity(pasteur);
fermEmpty.addSpannedActivity(cooling);
storFill.addSpannedActivity(cooling);
```



The `SetActivitiesSFP` function fully defines all of these constraints.

Tanks as storage units

Until now we made no differences between tanks. In fact, the modeling of preparation tanks and fermentation tanks as unary resources is sufficient because we don't model the intermediates they store. For the storage tanks however, we want to monitor the tank level over time as they supply the filling lines. So we must use a new concept called the *storage unit*. So within a loop of the function `SetStorageUnitsSFP` is this code:

```
IloMSStorageUnit suStorageTank = model.newStorageUnit();
suStoragetank.setName("Storage Tank" + (i + 1));
suStorageTank.setQuantityMax(35);
```

We must state which materials can be stored in the tank, which in this case are the three white masses:

```
suStorageTank.addStorableMaterial(bifidus);
suStorageTank.addStorableMaterial(bulgaricus);
suStorageTank.addStorableMaterial(bulgaricusLight);
```

We specify the maximum capacity:

```
suStorageTank.setQuantityMax(35);
```

We must link the storage unit to the corresponding resource:

```
suStorageTanks.setResource(storageTanks);
```

Production, consumption and destination

We must now define the storage unit to which the semi-finished product is produced. In the function `SetProductionsSFP` we create material production on the semi-finished recipe and note that one batch of semi-finished product may be consumed by many batches of finished products, hence the maximum number of outgoing pegging arcs is unlimited. First this:

```
String storageUnit = "suStorageTanks";
```

Then in a loop:

```
   IloMSMaterialProduction matProduction = model.newMaterialProduction(mode,
model.getMaterialByIdentifier(matIdSFP[m]), 1);
   matProduction.setName("Production of " + matIdSFP[m] + " on " + mode.
getResourceConstraint(0).getResource().getName());
   matProduction.setContinuous(true);
   matProduction.setMaxNumberOfPeggingArcs(IloMSConstants.IntPlusInfinity);
   matProduction.setStorageActivity(storageAct);
   matProduction.setStorageUnit(model.getStorageUnitByIdentifier(storageUnit)
);
```

Note that we want an order of finished product to be able to consume from one tank among several, to avoid creating as many modes as there are combinations of tank origin and packaging lines. We enabled this process by grouping storage tanks together into a super resource in *Representing process equipment*, then linking the super resource to the storage unit. .

The recipes of finished products must also take into account the semi-finished material required and the storage unit that supplies the material. Then we express the consumption of semi-finished product by final product at the rate of one-half ton of white mass (such as bifidus) to produce one pallet of finished product (such as bifidus lemon). A loop iterates over the flavors and the finished product activity identifiers to populate the consumption that needs to be modeled. The limit on the pegging arcs is to ensure that each finished product consumes from only one of the available tanks in the super resource. The process is continuous.

```
IloMSMaterialProduction matConsumption = model.newMaterialProduction(mode,
model.getMaterialByIdentifier(matIdSFP[m]),-0.5);
matConsumption.setName("Consumption of " + matIdSFP[m] + "to produce " +
matIdSFP[m] + " " + flavors[f]);
matConsumption.setContinuous(true);
matConsumption.setMaxNumberOfPeggingArcs(1);
matConsumption.setStorageUnit(model.getStorageUnitByIdentifier(storageUnit));
```

Consumption is expressed as a negative production.

Cleaning policy

We have already created a cleaning recipe, activities, and constraints for the finished product lines. Similar code applies to the production resources, although the cleaning rules are different. The preparation, fermentation, and storage tanks all have the same cleanup rule, and so are grouped together in the string `identifiersTanks`. This code from the `SetModesCleaning` function also links each tank to a CIP line.

```
int maxTimeBeforeCleanupTank = 48 * 60; // Tanks must be cleaned at least every
 48 hours (48*60)
for (int r = 0; r < identifiersTanks.length; r++)
{
  IloMSResource res = model.getResourceByIdentifier(identifiersTanks[r]);
  res.setCleanupRecipe(cleanupRecipeProcess);
  res.setMaxTimeBeforeCleanup(maxTimeBeforeCleanupTank);
  //Define modes
  IloMSMode mode = model.newMode(cleaningActivityProcess);
  mode.setName("cleaning " + res.getName());
  model.newResourceConstraint(mode, res, 1, true);
  model.newResourceConstraint(mode, cipStation, 1, false);
  // preparation tank 1 is linked to cipl line 1 etc.
  model.newResourceConstraint(mode, model.getResourceByIdentifier
(identifiersCipRes[r % identifiersCipRes.length]), 1, false);
  mode.setVariableCost(10000);
  mode.setFixedProcessingTime(180);
}
```

The recipe `cleanupRecipeProcess` is created earlier in the `SetRecipesCleaning` function.

```
IloMSRecipe cleanupRecipeProcess = model.newRecipe();
```

The pasto and the cooler have another set of cleaning rules, so we identify them together as `PastoCool` and set the following guidelines:

```
int maxNumberOfBatchesBeforeCleanupPastoCool = 3;
int maxIdleTimeBeforeCleanupPastoCool = 8 * 60;
```

Then iterating over `PastoCool`, these resources are assigned a cleanup recipe and the new cleanup rules.

```
 res.setCleanupRecipe(cleanupRecipeProcess);
 res.setMaxNumberOfBatchesBeforeCleanup
(maxNumberOfBatchesBeforeCleanupPastoCool);
 res.setMaxIdleTimeBeforeCleanup(maxIdleTimeBeforeCleanupPastoCool);
```

Calendars and breaks

A maintenance activity is planned for the first half day on cooler 1.

```
IloMSCalendar calendarCooler1 = model.newCalendar();
IloMSCalendarInterval maintenance =
  model.newCalendarInterval(calendarCooler1,0, 770);
maintenance.setBreak(true);
maintenance.setName("Maintenance");
cooler1.setCalendar(calendarCooler1);
```

That concludes our description of modeling a yogurt factory using the PPO Java™ API.

Reference Documentation

Contains information regarding customization and extension of Plant PowerOps, and other advanced information.

In this section

Customizing and Extending PPO
Describes customization of Plant PowerOps and implementation of plug-ins to extend and configure PPO.

Using PPO with Microsoft products
This section describes how to build and run examples delivered with Plant PowerOps when you use Microsoft® Visual C++ .NET and Microsoft Windows® XP. Included are instructions on creating a project and linking the target with PPO.

Entity Relationship Diagrams
Presents the available Entity Relationship Diagrams (ERDs).

Universal Modeling Language diagrams
Presents the available UML diagrams.

Date and time display
Describes how time and dates are displayed in C++ and Java™ output and in the Plant PowerOps GUI. Includes a list of all supported time zones.

Customizing and Extending PPO

Describes customization of Plant PowerOps and implementation of plug-ins to extend and configure PPO.

In this section

Authentication and access rights

This section describes how to setup authentication and access rights to the PPO GUI in order to control data access.

Customizing report generation with Tableau

Describes the use of Tableau within PPO.

GUI extension mechanism

This section describes how to extend and customize aspects of the PPO GUI.

Plan view customization

It is possible to add or remove plan views, rearrange the layout of a view, and select a new default layout for a view.

Customizing views, menus, and toolbars

You can add new or remove existing views, panels, table layouts, menu items or toolbar icons.

Engine optimizer extensions

An optimizer class defines how the data model is solved; it is possible to change this optimization class from the PPO default.

Configuring the data views
The data views include the **Master Data** and **Transactional Data** views which consist of tables that contain the model data and generated plan data. You can add, remove or modify data tables per your needs using the techniques of this section.

Database customization
This describes customizations that you can make regarding database usage in PPO. The list of supported databases is available at *Database usage and connectivity*.

Options of the PlantPowerOps executable file
How to display the DOS window and decrease GUI memory usage while running PPO.

Authentication and access rights

This section describes how to setup authentication and access rights to the PPO GUI in order to control data access.

In this section

User role determines access
In PPO access rights to GUI data are defined by one of three user roles.

Adding a login panel to PPO
You can configure PPO to open a login panel when started.

Advanced configuration of roles and access rights
Describes how to further tailor access to PPO data.

General security considerations
Protection of configuration files.

User role determines access

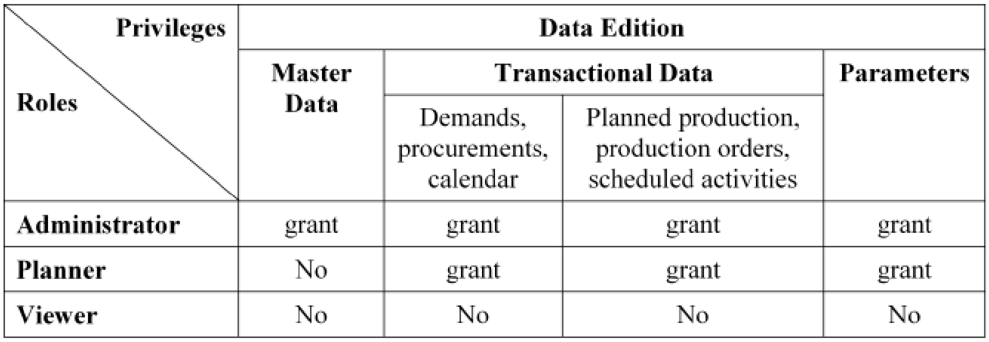
In PPO access rights to data through the GUI are defined by the user role. PPO determines the role of a user with a “login” step. A user can have only one role. The user login and password information can be stored in one of the following repositories:

- Microsoft® Active Directory
- Other LDAP directories
- XML file

By default PPO manages three roles:

- **Administrator**: This role has full access to all PPO features.
- **Planner**: This role is allowed to modify transactional data and run the PPO engine.
- **Viewer**: This role can only view PPO data.

The following table summarizes these roles.



Privileges / Roles	Data Edition			
	Master Data	**Transactional Data**		**Parameters**
		Demands, procurements, calendar	Planned production, production orders, scheduled activities	
Administrator	grant	grant	grant	grant
Planner	No	grant	grant	grant
Viewer	No	No	No	No

Adding a login panel to PPO

You can configure PPO to open a login panel when started. To activate this login panel you have to edit the file `login.xml` located in the directory `<PPOInstallDirectory>/data/gui/` and uncomment one of the three xml elements to choose the login method. These login methods are described below.

Using Microsoft Active Directory

To make PPO use Active Directory for user authentication you need to uncomment the following XML element in `login.xml`:

```
<page classname="ilog.plant.gui.login.ActiveDirectoryLoginPage">
      <host value="ldap://servername.domain.domainending"/>
      <port value="389"/>
      <domain value="domain.domainending"/>
      <root value="cn=Users,dc=domain,dc=domainending"/>
      <timeout value="2000"/>
 </page>
```

In the `host` and `port` tags you need to set the host name (or the IP address) where Active Directory is running and the port number (by default Active Directory uses port number 389). The `domain` tag should contain the domain name to which PPO users belong. The `root` tag is the Active Directory path where user entries are declared. After authenticating a user, PPO tries to retrieve its role by reading the `memberOf` attribute of the user.

You need to create PPO roles as groups of users in MS Active Directory. PPO expects to have the following groups with the following names:

♦ `PPO Admin` for administrators

♦ `PPO Planners` for planners

♦ `PPO Viewers` for viewers

PPO users must belong to one of these groups. Note that users of these groups must have the right to read their attributes stored in the Active Directory (at least the `memberOf` attribute).

Note also that some of the default values used by PPO for LDAP and Active Directory are located in `ldap.properties` file located in `<PPOInstallDirectory>/data`.

Using other LDAP directories

To use other LDAPs for implementation of PPO authentication, you need to uncomment the following XML element in `login.xml` file:

```
<page classname="ilog.powerops.logon.login.ldap.LDAPLoginPage">
      <host value="ldap://servername.domain.domainending"/>
      <port value="10389"/>
      <root value="dc=domain,dc=domainending"/>
      <timeout value="2000"/>
 </page>
```

The `host` and `port` tags need to be set to the host name (or the IP address) where the LDAP server is running. The `root` tag should contain the path where user entries are declared in the directory. By default, PPO uses the attribute `uid` to identify a user and the attribute `roleoccupant` to identify the user role. By default, the `roleoccupant` attribute must have one of the following values:

- ♦ `administrator`
- ♦ `planner`
- ♦ `viewer`

Note that some of the default values used by PPO for LDAP and Active Directory are located in `ldap.properties` file located in `<PPOInstallDirectory>/data`.

Using XML file for login and password

With this approach, user login and password information is stored in an XML file. You need to uncomment the following XML element in `login.xml` file:

The path tag must contain the relative or absolute path to the XML file. This file must have the following structure:

```
<Users xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="users.xsd">
      <User name="admin" password="O2HAFxib2rM=" role="administrator"/>
      <User name="planner" password="8VARVGJPrZ8=" role="planner"/>
      <User name="viewer" password="JdYC4C6SKsM=" role="viewer"/>
 </Users>
```

The name attribute is the login of the user, the password value is encrypted and the role represents the user role. To encrypt user password you can user the executable `<PPOInstallDirectory>/bin/Plant_Encrypt.bat`.

Advanced configuration of roles and access rights

Users access rights are defined in the configuration file `roleGuiAuthorizations.xml` that you can find in `<PPOInstallDirectory>/data`. You can change the actions granted to a given role by modifying this file. The complete list of PPO GUI actions can be found in the file `guiCommands.xml` located in `ilog/plant/gui/resources` in `plantgui.jar`. In the file `roleGuiAuthorizations.xml` you can add also new roles. New roles are taken into account automatically by PPO for authentication when using XML file and LDAP directory. For Microsoft® Active Directory you need to write your own login page to take into account new roles. By default for Active Directory PPO manages the default roles (administrator, planner and viewer). To write a login page you have to implement a Java™ class that extends `ilog.powerops.logon.login.LoginPage` and declare it in the `login.xml` file.

Fine grain access rights for PPO table views

You can define access rights on the columns of PPO table views based on the user role. Columns are identified by the name of the view (`MaterialView`, `ResourceView`,...) followed by '.' and the name of the column (`DaysOfSupplyTargetMax`, `DemandVariability`,...). You can find all views and column names in the XML configuration files in `<PPOInstallDirectory>/data/gui/table`. For instance, to prevent the planner role from changing the batch size of production order you have to add the following element in the `guiCommands.xml` file:

```
<role name="planner">
      <add>
          -
      </add>
      <remove>
          -
          <command>ProductionOrderView.BatchSize</command>
      </remove>
 </role>
```

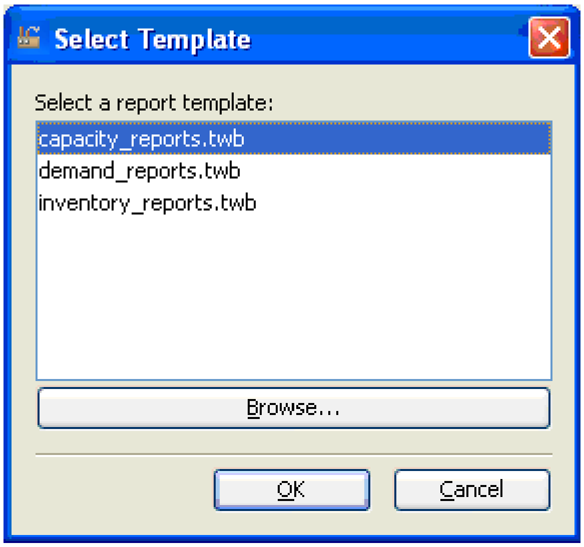
General security considerations

Some configuration files like `login.xml` and `guiCommands.xml` are important for security and the data manipulation of PPO. PPO looks for these files in its classpath. To protect them from malicious modification you can put these files in a directory and restrict the access rights by the operating system (only authorized users can change these files) or you can put them in a signed jar file (using `java jarsigner`). If the jar file has been altered, PPO will not run. If you want to increase the security of your PPO deployment, you can check the signature of this jar file in a plug-in.

Customizing report generation with Tableau

PPO uses Tableau to generate and display reports. Tableau is integrated as a plug-in; for Tableau documentation please go to *http://www.tableausoftware.com/* or type **F1** when Tableau starts. Note that Tableau is enabled for Unicode and compatible with data stored in any language; however the user interface and supporting documentation are available in English only.

By default, PPO is shipped with three report templates, as shown in this image:



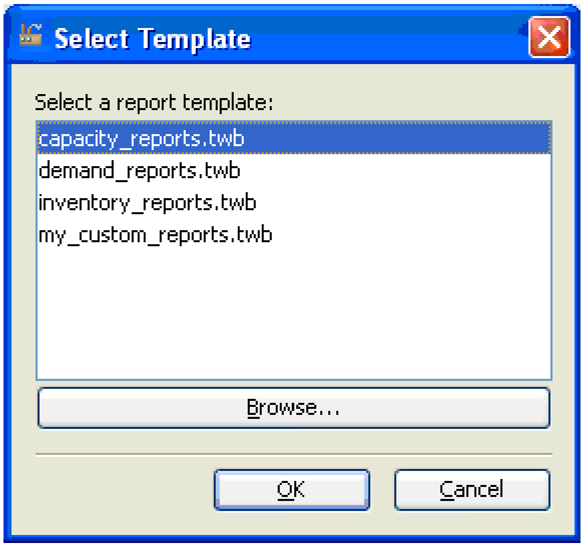
Select a report template:

capacity_reports.twb
demand_reports.twb
inventory_reports.twb

You can add your own templates to this list. To add a template:

1. Open one of the default templates to have the structure of data exported by PPO.
2. Modify the template as you like and save it in `<PPO_INSTALL_DIR>/plugins/tableau/data`.

The next time you try to generate a report, the new template appears in the **Select Template** list (`my_custom_reports.twb`):



Select a report template:

capacity_reports.twb
demand_reports.twb
inventory_reports.twb
my_custom_reports.twb

If you save your template to a different location than the one specified above, use the **Browse...** button to locate it.

Extending report data schema

If you want to export custom data to Tableau for report generating you need to extend the report data schema. PPO uses its standard mechanism to extend the data schema of the report (see *Customizing PPO data model tables*).

To extend the data model by adding a custom table you need to modify the Microsoft® Access® schema file in `<PPOInstalldirectory>/plugins/tableau/resources/ReportingSchema.mdb` and adapt the mapping file `<PPOInstallDirectory>/plugins/tableasu/resources/ReportingTables.xml` as described in *Adding custom tables*.

Finally, to populate the custom table when exporting the data to Tableau you need to write a listener and declare it in `<PPOInstalldirectory>/plugins/tableau/resources/reporting.properties` in the entry: `Reporting.DBWriterListenerClass=MyCustomTableListener`

Your listener class must implement the interface `ilog.plant.persistence.writer.DBWriterListener`. Your listener will be called when PPO exports data to Tableau.

GUI extension mechanism

This section describes how to extend and customize aspects of the PPO GUI.

In this section

Overview
Extensions to the GUI are made through bundles of code and data called plug-ins.

Format of plug-in manifest files - plugin.xml
Describes the elements of a plug-in file.

Extensions and extension points
Extensions are code bundles that are executed when PPO is performing a specific task.

Specific customizations
Describes exceptions to the use of extension points.

Overview of extension points
Extensions can be applied to the GUI Plan views or at the application level.

Overview

The PPO GUI extension mechanism is structured around the concept of plug-ins. Plug-ins are bundles of code and/or data that contribute extensions to the PPO application. Writing a PPO extension involves writing a new PPO plug-in or modifying an existing one.

One plug-in can contribute to several PPO extensions. For example, the same plug-in can customize the main menu of the application and change the optimizer used for solving data.

One plug-in corresponds to one subdirectory of the `${plant-gui.home}/plugins` directory. To be identified as a plug-in, this directory must also contain a file named `plugin.xml`. This file is named the manifest file of the plug-in and completely describes the plug-in.

Format of plug-in manifest files - plugin.xml

A manifest file lists all the extensions provided with the plug-in as well as all the resources to be used by these extensions, such as jar files or localization files.

A manifest file is an XML file with a structure defined as follows:

Manifest file structure

Elements	Attributes	Description
<plugin>		Manifest file root element
	name	Specify a name to identify the plug-in. This name can be used by other plug-ins which define dependencies with this plug-in.
	provider-name (optional)	Provide a name for the provider of the plug-in. This name should be a key to a string resource defined as a property file of the plug-in.
	classname (optional)	Provide the class of the plug-in object to instantiate when installing the plug-in. This class must inherit from `ilog.powerops.app.Plugin`. The plug-in object is automatically invoked when the plug-in is installed and uninstalled. See the documentation of the `Plugin` class for more details.
<bundles> (optional)		Provide all the property files to be used by the plug-in.
<bundle> (*)		Declare a property file to include with the plug-in. The property file must be located at the root directory of the plug-in. The base name of the property file is given with the text data of this element. For an example, see the plug-in example `.jar` file.
<runtime> (optional)		Provide all the libraries to be used by the plug-in.
<library> (*)		Include a jar file to be used by the plug-in's classloader when loading Java™ classes declared in this manifest file. For

Elements	Attributes	Description
		example, the plug-in classloader is used when loading the plug-in installer class declared in the `<plugin>` root element.
	name	Specify the file name of the jar file to include. For an example, see the plug-in example `.jar` file.
<requires>		Declare the list of plug-ins this plug-in depends on. This declaration ensures that plug-ins listed here will be loaded before this plug-in.
<import> (*)		Declare one plug-in this plug-in depends on.
	id	The value of this attribute is the name of the plug-in this plug-in depends on. Plug-in names are specified with the `name` attribute of `<plugin>` elements.
<extension-point> (*)(optional)		Declares a new extension point to the application. See *Extensions and extension points*.
	id	Unique identifier to identify the extension point. This identifier is referred to when declaring extensions to this extension point definition, as follows: `<extension point="\<id>"...>`
	name	This name is used to identify this plug-in when displaying messages that involve the plug-in.
	classname	The class of the extension point to associate with this extension point declaration. This class must inherit from the class `ilog.powerops.app.ExtensionPoint`.
	extensionClassname	The default class for extensions that are associated with this extension point. This class can be overridden for each extension declaration as follows: `<extension classname="MyOwnExtensionClass">`
<extension> (*)		Defines a new extension to the application. See *Extensions and extension points*.
	point	Specifies which extension-point this extension is relative to. The value of this attribute must correspond to the `id` specified in an extension point declaration as follows: `<extension-point id="..."/>`.
	classname *(optional)*	Specifies the class to instantiate this extension from. This declaration overrides the extension class given when declaring the extension point of this extension.
	name	Identifies this extension among other extensions that apply to the same extension point.

Elements	Attributes	Description
		This name can be referred to in other extension declarations to remove this extension declaration, with the `remove -extension` attribute.
	remove-extension *(optional)*	Removes the extension previously defined with the same extension point as defined by the `point` attribute and whose name corresponds to the value of this attribute. If this attribute is specified, attributes `classname` and `name` are not read.

Extensions and extension points

The main purpose of a manifest file is to declare *extensions* to the application. An *extension* is a bundle of XML and Java™ code that is automatically executed or used when the application is performing specific tasks. For example, when the application is initializing its main menu, it will execute all menu completions declared in extensions relative to menu customization. Such extensible tasks are named extension points.

Extension points are declared with the `<extension-point>` declaration in the plug-in manifest file. For each extension point declaration, the GUI instantiates an extension point object whose class is specified in the extension point declaration. This class must inherit from the `ilog.powerops.app.ExtensionPoint` class.

Extension point instances can be retrieved from the application with the method `AbstractApplication.getExtensionPoint(String id)` where the `id` parameter must correspond to the `id` attribute value given in the `<extension-point>` declaration of the plug-in.

An extension point instance is responsible for:

- ♦ Reading the content of the `<extension-point>` declaration (method `ExtensionPoint.read(Element element, FormReaderServices services)`). Note that apart from the attribute documented for this element, the format of the `<extension-point>` element is specific to each extension point.
- ♦ Reading the declarations of the extensions that refer to this extension point (method `ExtensionPoint.readExtension(Object extension, Element element, ResourceResolver resourceResolver, FormReaderServices services)`).
- ♦ Providing the collection of extensions associated with the extension point (method `ExtensionPoint.getExtensions()`).

As for extension points, extensions are declared in the manifest file of plug-ins, with the `<extension>` element. One extension object is instantiated for each `<extension>` declaration. The default class for this instance is defined in the `<extension-point>` declaration and can be overridden for each `<extension>` declaration. Extension classes do not have to implement or inherit from a specific class. They need only be provided with a default constructor.

Specific customizations

There are two exceptions to the use of extension points for extending an application. In these two exceptions, the extensions must be read when the application bootstraps and before the extension points are initialized.

These extensions are:

♦ changing the splash window of the application - the panel that pops up during the initialization phase of the application

♦ changing the name, the icon, and the title of the application.

These two extensions are defined in the manifest file as follows:

Specific customizations

Elements	Attribute	Description
<plugin>		
<application> *(optional)*		Override default parameter of the application.
	name *(optional)*	Changes the name of the application. This name is used to construct the path to user home settings on the Microsoft® Windows® platform. On this platform, the path is defined as follows: `%HOMEDRIVE%%HOMEPATH%\Application Data\ILOG\<application name>`
	title *(optional)*	Changes the title of the application main frame. The value of this attribute should be a key to a string resource defined in a property file declared for this plug-in (see `<bundles>` description above).
<icon> *(optional)*		Changes the icon for the PPO application.
<file>		The text data of this element defines the name of the image file. The file must be located at the root of the plug-in directory. For example, `<file>mycompany.gif</file>` looks for the `mycompany.gif` image file.
<splashWindow> *(optional)*		Changes the default splash window that pops up when the PPO application is launching.
	classname *(optional)*	The classname of the panel that pops up. If this attribute is not specified, a default panel is instantiated which is filled with the image of the splash window.
<file>		Locates the image to display in the splash window. The text data of this element contains the name of the file. The file must be at the root of the plug-in directory. For example, `<file>splash.gif</file>` displays the `splash.gif` image in the splash window.

Overview of extension points

Extensions can be applied to the GUI Plan views or at the application level.

In this section

Customization of Plan views
Describes extensions to the Plan views.

Application level extensions
Describes extensions to menu or toolbars.

Customization of Plan views

1. Plan view (`extension-point=PlanViewType`)

   Add a new view type to display a plan. Default view types are the **Data**, **Calendars**, **KPIs Summary**, **Stock Summary**, **Stock Coverage**, **Stock Event**, **Planning Sheet**, **Planning Workload**, **Workload Table**, and **Resources Gantt**. See *Plan view customization*.

   This extension point also lets you remove or modify predefined view types.

2. Plan view layout configuration (`extension-point= PlanViewContainer.Configuration`)

   See *Plan view customization*.

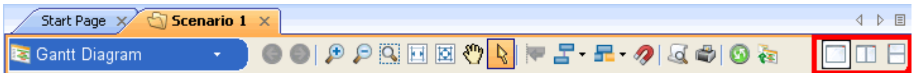
   Lets you provide a new layout to configure plan views in the same plan tab. By default, three configurations are provided:

   ♦ `single`. One plan view is contained in the plan tab

   ♦ `hsplit`. A plan tab is horizontally split to contain two plan views

   ♦ `vsplit`. A plan tab is vertically split to contain two plan views.

   The application user can navigate between these configurations by selecting configuration buttons in the upper right corner of the plan tab:

3. Default plan view layout (`extension-point= PlanViewContainer.Defaults`)

   The default for PPO is one view. If there is a scheduling solution, the view is the **Resources Gantt**. If there is only a planning solution, the view is the **Planning Sheet**. See *Plan view customization*.

Application level extensions

Menu customization (`extension-point= MenuCustomization`)

Used to insert, remove or modify menu items in the main menu or toolbars of the application. This extension let you also specify Java™ code to call when the user activates this new menu item. See *Insert a new item in the menu and a new icon button in the main toolbar* and *Remove menu item and button from the main toolbar*.

Plan view customization

It is possible to add or remove plan views, rearrange the layout of a view, and select a new default layout for a view.

In this section

Overview
An overview of the extension points.

PlanViewTypes extension point
This extension point is used to add new plan view types or remove predefined types.

PlanViewContainer.Configuration extension point
Use this section to change the layout of a plan view.

PlanViewContainer.Defaults extension point
Use this section to define and declare the default view layout.

Plan view extension example
Files are included in the distribution to help you understand how to configure the data views.

Overview

In this section you will learn about:

- ♦ *PlanViewTypes extension point*
- ♦ *PlanViewContainer.Configuration extension point*
- ♦ *PlanViewContainer.Defaults extension point*
- ♦ *Plan view extension example*

The PPO GUI provides three extension points to customize the visualization of plans within plan views. It is possible to provide the GUI with new plan view types or remove predefined ones (`extension-point="PlanViewType"`), provide new layouts for arranging views of a same plan into a plan view container (`extension-point="PlanViewContainer.Configuration"`), and select the layout to use as the default layout (`extension-point="PlanViewContainer.Defaults"`).

PlanViewTypes extension point

This extension point is used to add new plan view types or remove predefined types.

In this section

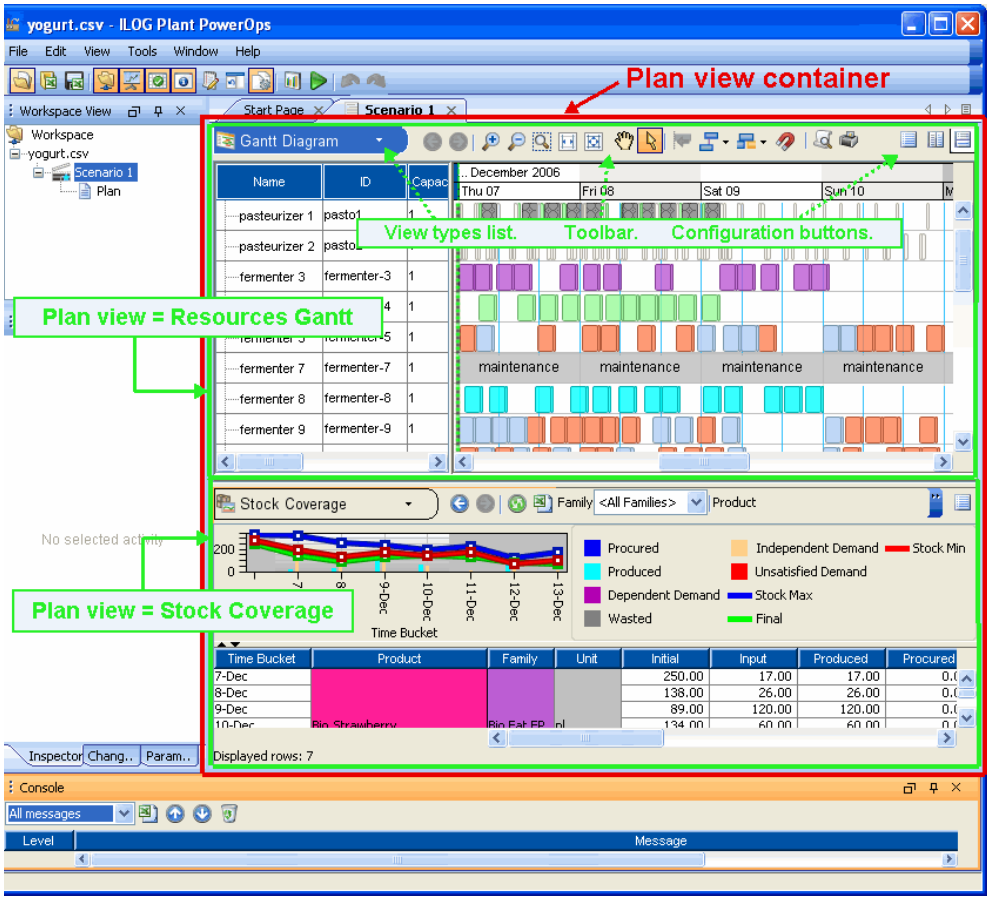
Overview
Defines terms you need to know.

Definition of a new plan view type
Describes the XML declaration and Java class necessary to change a plan view type.

Remove a predefined plan view type
The extension declaration must be removed.

Overview

The following diagram presents the terms used in this section:



Time Bucket	Product	Family	Unit	Initial	Input	Produced	Procured
7-Dec				250.00	17.00	17.00	0.(
8-Dec				138.00	26.00	26.00	0.(
9-Dec				89.00	120.00	120.00	0.(
10-Dec	Bio Strawberry	Bio Fat FP	nl	134.00	60.00	60.00	0.(

The Plan view container in this example holds two plan views: **Resources Gantt** and **Stock Coverage**. Each Plan view consists of the View type list, the toolbar, the configuration buttons, and the view itself.

Default views for a plan are:

- Data
- Calendars
- KPIs Summary
- Stock Summary

- Stock Coverage
- Stock Event
- Planning Sheet
- Planning Workload
- Workload Table
- Resources Gantt

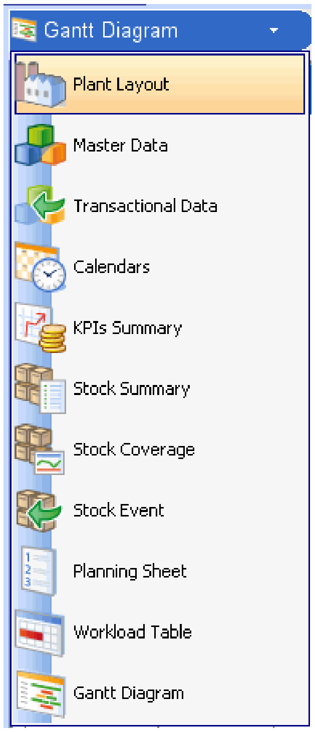
These predefined view types are declared to the PPO GUI using the extension point `PlanViewType`. These declarations are located in the configuration file `config.xml`, in the directory `<install directory>\data\gui`.

It is possible to use the same extension point for defining new plan view types, or to remove predefined ones.

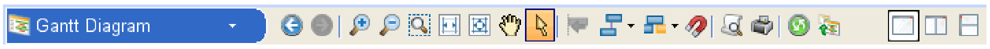
Definition of a new plan view type

As with many extension points, the `PlanViewType` extension point is a bundle of an XML declaration and a Java™ class. The XML declaration provides the GUI with the following information:

- How to display this new plan view type in the plan view type list. This is the list the pops up when activating the combo at the top left side of a plan view. Here is a screen shot for this combo:



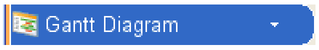
- Provide the toolbar associated with each plan view of this type. A screen shot of the toolbar of the **Resources Gantt** view is:

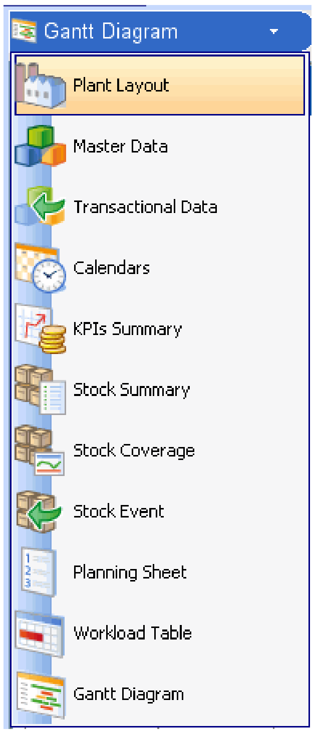


The Java class to implement for this extension point is the plan view, the component to visualize and edit a plan.

Plan view classes must implement the interface `ilog.powerops.workspace.PlanView`. See the *Plant PowerOps Java Reference Manual* for more information.

Plan view extension format

Elements	Attributes	Description
<plugin>		Manifest file root element
<extension point="PlanViewType">		For defining a new plan view type
	name	The name of this extension. This name can be referenced by a plug-in which wants to remove this extension.
	javaClass	The class of the plan view. This class must implement the interface `ilog.powerops.workspace.PlanView`. See section below for more details on the plan view to implement.
	title	The title to display for this plan view type. This title is visible both in the top left corner of a plan view container or in the drop-down list that lists all plan view types. This title should be a key to a string resource associated with the plug-in.
	shortDescription	A short description of this plan view type. Is used for tooltips associated with this plan view type. This description should be a key to a string resource associated with the plug-in.
	longDescription	A long description of this plan view type. Is displayed in the status bar of the application when this plan view type is selected in the drop-down list of plan view types. This description should be a key to a string resource associated with the plug-in.
<icon>		The 16x16 icon image to display with the title of this plan view type in the top left corner of the plan view container. Here is a screen shot of this display for the **Resources Gantt** view:  The text data of this element contains the filename of the image file, as for example `<icon>factory16.gif</icon>`. The image should be located at the root directory of the plug-in that declares this extension.
<largeIcon>		The 32x32 icon image to display in the drop-down list of plan view types. Here is a screen shot of the default PPO GUI drop-down list:

Elements	Attributes	Description
		 Gantt Diagram Plant Layout Master Data Transactional Data Calendars KPIs Summary Stock Summary Stock Coverage Stock Event Planning Sheet Workload Table Gantt Diagram The text data of this element contains the filename of the image file, as for example `<largeIcon>factory32.gif</largeIcon>`. The image should be located at the root directory of the plug-in that declares this extension.
<toolbar>		Defines the toolbar associated with plan views of this type. Buttons and separators of this toolbar are declared with child elements of this element, using the same format as for inserting button and separator items in <insert> elements in menu customization.

<popups>		Declares a list of pop-ups to be used by the plan view. A plan view can
		
		access these popups with the code:
		
		`PlanView view = ...`
		
		`JPopupMenu popup = view.getViewType().getPopupMenu( <name of the popup menu>);`

<popupMenu>		Defines a popup to be used by plan views associated with this type. Component items that compose this popup menu – such as menu items or separators - are declared with child

Elements	Attributes	Description
		elements of this element, using the same format as for inserting component items with the <insert> elements in menu customizations.
	name	The name to identify the popup. See <popups> for the use of this name by code.

Remove a predefined plan view type

Removing a predefined plan view type is done by removing the extension that declares this plan view type. The following XML code to remove the **Resources Gantt** plan view type must be inserted in the plug-in manifest file:

```
<plugin>
   …
   <extension point="PlanViewType" remove-extension="Resources"/>
   …
</plugin>
```

The "Resources" value corresponds to the name given to the extension declaration of the resource type, with the id attribute. Extension IDs for the predefined views are:

Extension IDs for Predefined Views

Plan view type	Extension Name
Data	Data
Calendars	Calendar
KPIs Summary	Summary
Stock Summary	StockSheet
Stock Coverage	StockCoverageSheet
Stock Event	StockEventSheet
Planning Sheet	PlanningSheet
Planning Workload	Shifts
Workload Table	WorkloadTable
Resources Gantt	Resources

PlanViewContainer.Configuration extension point

Use this section to change the layout of a plan view.

In this section

Overview
You can add to the three standard configurations of displaying a view.

Add a new plan view configuration
The mechanism of adding a new view configuration.

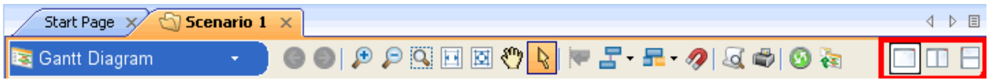
Implement a configuration factory class (optional)
Used to create a new container.

Implement a configuration
To position the plan views within a container.

Remove a plan view configuration
The code to remove a plan view.

Overview

The PPO GUI allows you to configure views of the same plan within a window, called the plan view container. These configurations are activated through the configuration buttons highlighted in the following figure:



The PPO GUI provides three configurations:

- Single: One plan view is displayed in a plan view container (default).
- Horizontal Split: Two plan views are horizontally split in a plan view container.
- Vertical Split: Two plan views are vertically split in a plan view container.

Add a new plan view configuration

Plan view configurations are provided with extensions to the extension point `PlanViewContainer.Configuration`.

The XML format for this extension is:

Plan view configuration extension format

Elements	Attributes	Description
<plugin>		Manifest file root element
<extension point="PlanViewContainer.Configuration">		For defining a new plan view configuration
	id	The `id` of this extension. This `id` can be referenced by a plug-in which wants to remove this configuration or to set this configuration as the default configuration (see `extension -point="PlanViewContainer.Defaults"`).
	classname *(optional)*	The class of the extension Java™ object. It must inherit from `ilog.powerops.workspace.PlanViewConfigurationExtensionPoint.ConfigurationFactory`. It is responsible for creating a configuration for each newly created plan view container.
	configurationClass	The class of the configurations to create for plan view containers. It must inherit from `ilog.powerops.workspace.PlanViewContainer.Configuration`. This is the class responsible for arranging plan views within a plan view container.
	icon	The filename of the image file to provide as an icon to the button associated with this configuration. The image file must be at the root of the plug-in directory that declares this extension.
	title	A title for this configuration. This title should be a key to a string resource associated with the plug-in.
	description	A long description for this configuration. This description should be a key to a string resource associated with the plug-in.
	tooltip	A tooltip for buttons associated with this configuration. This tooltip should be a key to a string resource associated with the plug-in.
<site> (*)		Configures a plan view site to install a plan view into. A site is the container that contains the plan view, its associated toolbar, and the drop-down list that allows for changing the plan view.

Elements	Attributes	Description
		The number of `<site>` elements declared in the `<extension>` elements determines exactly the number of plan view sites the configuration can host. For example, the vertical split configuration defines two sites which are vertically split. Its declaration contains two `<site>` elements.
	defaultType (optional)	Provides the `id` of the plan view type to display by default in this place holder.
	typeListVisible (optional)	Determines whether the drop-down list for changing the type of plan view should be visible or hidden. If the value of this attribute equals `"false"`, the drop-down list is hidden. Otherwise, the list is shown.

Implement a configuration factory class (optional)

A configuration factory is responsible for instantiating and initializing a new `ilog.powerops.workspace.PlanViewContainer.Configuration` for each plan view container. The configuration factory class is specified with the classname attribute of the `PlanViewContainer.Configuration` extension and must inherit from the class `ilog.powerops.workspace.PlanViewConfigurationExtensionPoint.ConfigurationFactory`.

This base class instantiates a default configuration factory if no `classname` attribute is specified for the extension. The default behavior of this class is to read the configuration class from the `configurationClass` attribute of the extension and to instantiate a configuration from this class for each newly created plan view container.

However, it is necessary to define your own configuration factory for reading additional parameters from the extension parameters. These parameters can then be given to the instantiated configurations from the method `initializeConfiguration(PlanViewContainer.Configuration configuration)`.

Implement a configuration

A configuration is responsible for arranging plan view sites (one view site contains one plan view) within a plan view container. The configuration class is specified from the `configurationClass` attribute of the extension.

Main methods to override are:

- `getRootContainer()`: The container that will contain all the plan view sites. This container fits all the area of the plan view container when this configuration is selected. For split configuration, this container is the `JSplitPane` that contains two plan view sites.

- `insertPlanViewSite(PlanViewSite site, PlanViewSite oldSite, int index)`: Insert a plan view site in the root container returned with the method `getRootContainer`. This method is invoked for each plan view site to install in the configuration.

Remove a plan view configuration

Removing a predefined plan view configuration consists of removing the extension that declares this plan view configuration. For example, the following XML code to remove the `hsplit` plan view configuration must be inserted in a plug-in manifest file:

```
<plugin>
   …
   <extension point="PlanViewContainer.Configuration"
              remove-extension="hsplit"/>
   …
</plugin>
```

PlanViewContainer.Defaults extension point

The extension point "`PlanViewContainer.Defaults`" determines the configuration to set by default to a plan view container that was just created.

The XML format for this extension is:

Plan view container default extension format

Elements	Attributes	Description
<plugin>		Manifest file root element
<extension point="PlanViewContainer.Defaults">		For changing the default plan view configuration
	id	The `id` of this extension. This `id` can be referenced by a plug-in which wants to remove this extension.
	defaultConfiguration	Specifies the `id` of the plan view configuration to use by default. The value of this attribute must correspond to the value of the `id` attribute specified in the `<extension point="PlanViewContainer. Configuration"…>` declaration of the plan view configuration.

	defaultPlanViewTypes (optional)	Specifies the default plan view types to select in the plan view sites installed in the configuration.
		
		The value of this attribute is a comma separated list of names, each name being the name of a plan view type.
		
		These specified plan view types do not override default plan view types which may be specified in the declaration of the configuration.
		
		For example, consider a configuration declaring two plan view sites, as follows:
		
		`<extension`
		
		`point="PlanViewContainer. Configuration"`
		
		`defaultConfiguration= "twoSites">`
		
		`<site defaultType="Resources"/ >`
		
		`<site/>`

Elements	Attributes	Description

		`</extension>`
		
		Now consider a `"PlanViewContainer.Defaults"` extension that declares two default types for this configuration as follows:
		
		`<extension`
		
		`point="PlanViewContainer. Defaults"`
		
		`defaultConfiguration= "twoSites"`
		
		`defaultPlanViewTypes= "Resources,Summary">`
		
		`</extension>`
		
		The default plan view types for this configuration will be **Resources Gantt** and **KPIs Summary**.

Plan view extension example

Files are included in the distribution to help you understand how to configure the data views.

In this section

Overview
A list of useful files.

How to run the example
Compile the plug-in first.

Overview

The Plant PowerOps distribution comes with a set of example files that demonstrate how to create a new plan view. This new plan view displays information about setup activities of the current plan.

The sample files are located in the directory `<InstallDirectory>\examples\plugins\addSetupView`. The sample files include the following:

- a `build.xml` file
- a `plugin.xml` file
- two `.xml` view description files:

  `data\gui\table\setupdataaccess.xml` and

  `setupdisplay.xml;`
- three Java™ files:

  `src\AddSetupViewPlugInInstaller.java;`

  `src\ilog\plant\gui\docview\IloMSSetupActivityPanel.java;`

  `src\ilog\plant\gui\model\table\data\IloMSSetupActivityDataProviderOutside.java;`
- Associated image and properties files

How to run the example

To run the example, you need to compile the plug-in first. In the current plug-in directory (`addViewSetup`) just do so by launching `ant` (no argument needed). The desired plug-in is automatically copied to the general plug-in directory in `plant`.

Customizing views, menus, and toolbars

You can add new or remove existing views, panels, table layouts, menu items or toolbar icons.

In this section

Adding new table layouts
How to add new user-selectable column layouts for table views.

Inserting a new panel view
Describes the code to insert a new view.

Insert a new panel and toolbar in an existing view
Describes the code necessary to insert a new panel.

Insert a new item in the menu and a new icon button in the main toolbar
Describes the code needed to add a new menu command or icon.

Remove menu item and button from the main toolbar
The code necessary to remove a path to an existing menu or icon.

Adding new table layouts

This section describes how to add new user-selectable table column layouts. This allows a PPO GUI user to change the order and appearance of columns in the tables of certain views like the **Planning Sheet** and **Workload** views.

PPO searches for file names of the type `<view_name>displayXXX.xml` in the default path `plant/data/gui/table` as well as in user-specified directories (described later). Any defined table layouts from these files are then enabled to appear in the **Layout** combo box of the view. Define the filename as an i18n tag in one of the message properties files.

For example, the **Planning Sheet** view has two xml display files:

♦ `planningsheetdisplay.xml`

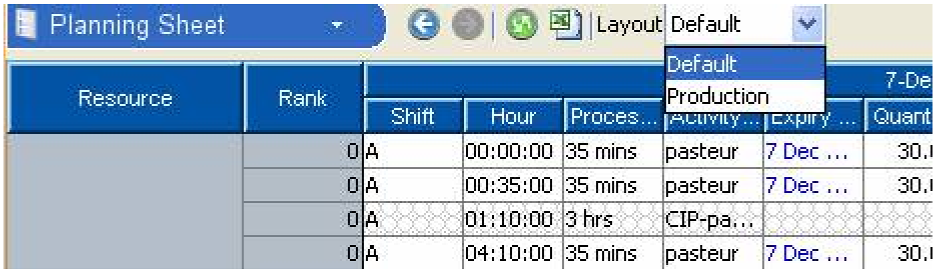
♦ `planningsheetdisplayproduction.xml`

There are two associated message property file entries:

♦ `planningsheetdisplay.xml=Default`

♦ `planningsheetdisplayproduction.xml=Production`

Which then allows the user to select the prefered layout, as shown in the following GUI image:



Resource	Rank	Shift	Hour	Proces...	Activity...	Expiry ...	Quant
	0	A	00:00:00	35 mins	pasteur	7 Dec ...	30.
	0	A	00:35:00	35 mins	pasteur	7 Dec ...	30.
	0	A	01:10:00	3 hrs	CIP-pa...		
	0	A	04:10:00	35 mins	pasteur	7 Dec ...	30.

To specify a user specific directory for your display files you need to call the static method `IloMSPlanningSheet.setOptionalDisplayPath(String path)` or `IloMSWorkloadPanel.setOptionalDisplayPath(String path)`.

Inserting a new panel view

This section provides an example of how to insert a new view to list all setup activities (if they exist) with their name, ID, start time, end time, setup cost and setup time.

1. Declare the structure of the new view in an `.xml` file, `setupDisplay.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<DatabaseEditor>
  <Views>
    <View id="SetupActivityView" label="AddSetupView.SetupActivities">
      <Columns>
       <Column id="Identifier" label="AddSetupView.identifier" width="94"

                  readonly="true" freeze="true"/>
        <Column id="Name" label="AddSetupView.name" width="100"
freeze="true"
                  readonly="true"/>
        <Column id="StartDate" label="AddSetupView.startTime" width="120"
                  readonly="true"/>
        <Column id="EndDate" label="AddSetupView.endtime" width="120"
                  readonly="true"/>
        <Column id="SetupCost" label="AddSetupView.setupCost" width="120"
                  readonly="true"/>
        <Column id="SetupTime" label="AddSetupView.setupTime" width="100"
                  readonly="true"/>
      </Columns>
    </View>
  </Views>
</DatabaseEditor>
```

2. Define the types and accessibility of fields in an `.xml` file, `setupDataaccess.xml`:

```
<DatabaseEditor>
  <Views>
    <View id="SetupActivityView" type="java"
provider="ilog.plant.gui.model.table.data.
IloMSSetupActivityDataProviderOutside
"
          rowtype="ilog.plant.IloMSScheduledActivity">
      <Columns>
        <Column id="Identifier" type="java.lang.String"
                  neededforcreation="true">
          <Get call="getIdentifier"/>
          <Set call="setIdentifier"/>
        </Column>
       <Column id="Name" type="java.lang.String" neededforcreation="true">

          <Get call="getName"/>
          <Set call="setName"/>
        </Column>
        <Column id="StartDate" type="java.util.Date">
          <Get call="getStartDate"/>
        </Column>
        <Column id="EndDate" type="java.util.Date">
          <Get call="getEndDate"/>
        </Column>
        <Column id="SetupCost" type="int">
          <Get call="getSetupCost"/>
        </Column>
        <Column id="SetupTime" type="int">
          <Get call="getSetupTime"/>
        </Column>

      </Columns>
    </View>
  </Views>
</DatabaseEditor>
```

3. Define the Java class managing the accessibility of fields, `IloMSSetupActivityDataProviderOutside.java`:

```

public class IloMSSetupActivityDataProviderOutside implements JavacallsView

{
…/…

  /**
   * Returns an iterator over a list of <code>IloMSScheduledActivity</code>.

   */
  public Iterator iterator()
  {
    ArrayList rowList = new ArrayList();
    IloMSSchedulingSolution solution = getSchedulingSolution();
    if (solution != null) {
      int nbObjects = solution.getNumberOfScheduledActivities();
      // Add the objects
      for (int i = 0; i < nbObjects; i++) {
        IloMSScheduledActivity object = solution.getScheduledActivity(i);
        if(object.isSetupActivity())
        {
                rowList.add(object);
        }
      }
    }
    return rowList.iterator();
  }

…/…

  /**
   * get the id of the IloMSScheduledActivity
   *
   * @param selected IloMSScheduledActivity
   * @return the id as string if success
   */
  public String getIdentifier(IloMSScheduledActivity activity)
  {
    try {
      return getAbstractActivity(activity).getIdentifier();
    } catch (Exception e) {
      ErrorHandler.Error(data.getPlantPlan().getApplication(), e);
      return null;
    }
  }

  /**
   * get the name of the IloMSScheduledActivity
   *
   * @param selected IloMSScheduledActivity
   * @return the name as string if success
   */
  public String getName(IloMSScheduledActivity activity)
  {
    try {
      return getAbstractActivity(activity).getName();
    } catch (Exception e) {
      ErrorHandler.Error(data.getPlantPlan().getApplication(), e);
      return null;
    }
  }
…/…
```

4. Define the Java class managing the new panel, `IloMSSetupActivityPanel.java`.
5. Declare this new panel in the plug-in manifest.

```
<plugin name="Plugin.addSetupView"
        provider-name="CompanyProvider"
        version="Plugin.Version"
        pluginInstallerJavaClass="AddSetupViewPlugInInstaller">
  <runtime>
    <library name="addSetupView.jar"/>
  </runtime>

  <settings>
    <file name="actions.xml"/>
  </settings>

  <extension point="PlanViewType"
             id="NewSetupView"
             name="SetupView"
             title="AddSetupView.Title"
             shortDescription="AddSetupView.Tooltip"
             longDescription="AddSetupView.LongDescription"
             javaClass="ilog.plant.gui.docview.IloMSSetupActivityPanel">
             <icon type="relativeToClass"
```

```
               classname="ilog.plant.gui.docview.IloMSSetupActivityPanel">

                ./resources/images/setupView16.gif</icon>
      <largeIcon type="relativeToClass"
             classname="ilog.plant.gui.docview.IloMSSetupActivityPanel">
             ./resources/images/setupView32.gif</largeIcon>
    <toolbar>
      <button actionCommand="Refresh"/>
      <button actionCommand="ExportToExcel"/>
      <button actionCommand="TotalSetupTime"/>
    </toolbar>
  </extension>

…/…
```

6. Declare the `TotalSetupTime` action. First, declare the new `TotalSetupTime` action in a new file `actions.xml`, located in the root directory of the plug-in. The `actions.xml` file declares the new action as follows:

```
<?xml version="1.0"?>
<appframe>
  <settings>
     <action actionCommand="TotalSetupTime"
      name="AddSetupView.action"
      tooltip="AddSetupView.action"
      description="TotalSetupTime"
      icon="./data/images/setupView16.gif">
    </action>
  </settings>
</appframe>
```

Resource strings referenced in this `.xml` file are defined in a property file named `addSetupView.properties`:

```
AddSetupView.action=Compute Total Setup Time
AddSetupView.computed=The Total Setup Time is
AddSetupView.notComputed=The Total Setup Time is not yet computed
…
```

7. The final step is to reference the action handler class in the plug-in installer. This is achieved as follows:

```
public class AddSetupViewPlugInInstaller
  extends Plugin
{
…/…

  public void install(Abstractapplication absApp)
  {
    super.install(absApp);
    System.err.print("installing");
    try {
   final IloMSPlantApplication plantApplication = (IloMSPlantApplication)
 absApp;
   pluginInfo = plantApplication.getServices().getString("plugin.exercise")
+

                     "-v"+
                     plantApplication.getServices().getString("plugin.
version");

System.err.println(pluginInfo + " plugin...");
absApp.getActionManager().addActionListener("TotalSetupTime", new
IlvSingleActionHandler("TotalSetupTime") {
public void actionPerformed(ActionEvent e) {
             JOptionPane.showMessageDialog(null,
getTotalSetupTimeInformation());
      }

      /**
       * get the formatted string
       * @return the formatted string
       */
      private String getTotalSetupTimeInformation() {
              String setupTimeInformation = plantApplication.
getApplicationContext().getString("AddSetupView.notComputed");
               if(plantApplication!=null) {
                       PlantPlan plan = (PlantPlan)plantApplication.
getActivePlan();
                       if(plan!=null) {
                              IloMSSchedulingSolution solution = plan.
getSchedulingSolution();
                              setupTimeInformation = plantApplication.
getApplicationContext().getString("AddSetupView.computed")

+computeTotalSetupTimeInformation(solution);
                    }
               }
               return setupTimeInformation;
      }

      /**
       * Compute the total setup time of all setup activities
       * @param solution
       * @return the total setup time as string
       */
       private String computeTotalSetupTimeInformation
(IloMSSchedulingSolution solution) {
               int totalSetupTime = 0;
        if (solution != null) {
          int nbObjects = solution.getNumberOfScheduledActivities();
          for (int i = 0; i < nbObjects; i++) {
            IloMSScheduledActivity object = solution.getScheduledActivity
(i);
            if(object.isSetupActivity())
            {
               totalSetupTime+=solution.getSetupTime(object.
getGeneratedActivity());
            }
          }
        }
                return String.valueOf(totalSetupTime);
    }

    }
    );

…/…

    System.out.println("New Setup View Plugin installed");
  }

…/…
}
```

Insert a new panel and toolbar in an existing view

This section provides an example of how to insert a new panel to list all existing setup activities with their name, ID, start time, end time, setup cost and setup time in the existing data view.

1. Declare the optional files (`setupDataaccess.xml` and `setupDisplay.xml`) which modify the content of the view as follows:

```
public class AddSetupViewPlugInInstaller extends Plugin
{
  public static final String DATA_ACCESS_FILE_NAME = "setupdataaccess.xml";

  public static final String DISPLAY_FILE_NAME = "setupdisplay.xml";
  public static final String DIRECTORY = "../plugins/addSetupView/data/gui/
table/";
     private String pluginInfo;
     public void install(AbstractApplication absApp) {
            super.install(absApp);
            System.err.print("installing ");
            try {
                 final IloMSPlantApplication plantApplication =
(IloMSPlantApplication) absApp;
                 pluginInfo = plantApplication.getServices().getString
("plugin.exercise")+
                                           "-v"+
                                           plantApplication.getServices()
.getString("plugin.version");
     System.err.println(pluginInfo + " plugin...");
..
     // installing the new table in the data view also
     IloMSDataPanel.SetOptionalDataFileName(DIRECTORY,
DATA_ACCESS_FILE_NAME, DISPLAY_FILE_NAME);
     System.err.println("Plugin installed");
    }
    catch (Exception e)
    {
     e.printStackTrace();
     }
        }
```

2. Declare the toolbar customization in the manifest file. The toolbar customization is declared in the manifest `.xml` file as follows:

```
<plugin>
<!--Declares the file containing actions associated with menu customizations
-->
…/…

  <!-- Add a new icon button in the data view toolbar -->
  <extension point="MenuCustomization">
    <menuCustomization>
      <barComponentID name="planViewBar[Data]"/>
      <insert path="History" after="true">
                      <separator/>
                   <button actionCommand="TotalSetupTime"/>
      </insert>
    </menuCustomization>
  </extension>

…/…
```

Insert a new item in the menu and a new icon button in the main toolbar

This section shows how to insert a new item `SetupTime` with its icon in the menu and a new button in the main toolbar of the application. When activated, the `SetupTime` action associated with these items displays a message with the sum of all setup times to the user. This button is in the new view created in the previous section.

1. Declare the action (as before).
2. Declare the menu and toolbar customization in the manifest file. The menu and toolbar customization is declared in the manifest XML file as follows:

```
<plugin>
<!--Declares the file containing actions associated with menu customizations-

->
…/…
  <!-- ADD A NEW MENU IN THE MENU tools after the solve menu -->
  <extension point="MenuCustomization">
    <menuCustomization>
      <barComponentID name="menu"/>
               <insert path="toolsMenu/SolveProblem" after="true">
                      <menuItem actionCommand="TotalSetupTime"/>
               </insert>
    </menuCustomization>
  </extension>

  <!-- ADD A NEW ICON BUTTON IN THE MAIN TOOL BAR after the solve button -->
  <extension point="MenuCustomization">
    <menuCustomization>
      <barComponentID name="toolbar[mainToolBar]"/>
      <insert path="CloseProblem" after="false">
                    <button actionCommand="TotalSetupTime"/>
                      <separator/>
      </insert>
    </menuCustomization>
  </extension>
```

Remove menu item and button from the main toolbar

This section shows how to remove an existing menu and button in the PPO GUI. The menu and toolbar customization is declared in the manifest XML file as follows:

```
  <plugin>
…/…

  <!-- REMOVE A MENU IN THE MENUS -->
  <extension point="MenuCustomization">
    <menuCustomization>
      <barComponentID name="menu"/>
               <remove path="toolsMenu/ShowKPIComparator"/>
    </menuCustomization>
  </extension>

  <!-- REMOVE A BUTTON IN THE MAIN TOOL BAR -->
  <extension point="MenuCustomization">
    <menuCustomization>
      <barComponentID name="toolbar[mainToolBar]"/>
               <remove path="ShowKPIComparator"/>
    </menuCustomization>
  </extension>
…/…
```

Engine optimizer extensions

An optimizer class defines how the data model is solved; it is possible to change this optimization class from the PPO default.

In this section

Overview
Implementation is through an optimizer class inheriting from `ilog.plant.IloMSEngineOptimizer`.

Write the engine optimizer factory class
Implementing the `IloMSOptimizerFactory` class.

Write the engine optimizer class to solve data
You can use the default engine optimizer or create your own.

Overview

In this section you will learn how to:

- ♦ *Write the engine optimizer factory class*
- ♦ *Write the engine optimizer class to solve data*

This section explains how to redefine how model data (including from a plan) is solved. These services must be implemented by an optimizer class inheriting from `ilog.plant.IloMSEngineOptimizer`.

For each of these services, an optimizer is created and deleted after the application dies. An optimizer factory is responsible for creating these optimizers each time it is necessary. An optimizer factory must inherit from `ilog.plant.IloMSOptimizerFactory` and is declared in the XML optimizer extension.

Write the engine optimizer factory class

Implementing the `IloMSOptimizerFactory` class.

In this section

Overview
Building the optimizers.

Write your own engine optimizer factory class
Declaring the new optimizer.

Overview

After declaring the optimizer extension in the `.xml` manifest file, you must implement an optimizer factory using the class `IloMSOptimizerFactory`. An optimizer factory is responsible for creating instances of `ilog.plant.IloMSEngineOptimizer`.

Use the factory method `IloMSOptimizerFactory.create` to build the optimizers.

```
public abstract IloMSOptimizerFactory create()
```

When no optimizer extensions are specified, the optimizer factory is automatically given the default factory used by the PPO GUI. For example, the default optimizer factory can be used by a custom engine optimizer for solving plans using different heuristics. The default optimizer factory is accessible in the optimizer factory class using the methods:

```
public static IloMSOptimizerFactory getFactory()
public static void setFactory(IloMSOptimizerFactory defaultFactory)
```

Write your own engine optimizer factory class

To replace the standard factory and optimizer, you can redefine the factory and declare your own. You need to inherit from `IloMSOptimizerFactory` and write the create method in which you create a new instance of the new optimizer. To set it inside PPO, you need to call the methods:

```
public static IloMSOptimizerFactory getFactory()
public static void setFactory(IloMSOptimizerFactory defaultFactory)
```

Write the engine optimizer class to solve data

You can use the default engine optimizer or create your own.

In this section

Overview
The four steps that involve the new optimizer.

Launch and monitor the solve process
Using the solve method.

Use the default engine optimizer
Methods to retrieve and solve using the default optimizer.

Create your own engine optimizer
The methods to create, retrieve, and extend the optimizer.

Stopping the solve process
How to stop a solve process.

Overview

Engine optimizers, implemented using the class `IloMSEngineOptimizer`, are responsible for producing plans (solutions). When the PPO GUI is asked for solving data, it involves the engine optimizer in the following steps:

1. Provides the optimizer with the engine parameters.
2. Launches the solve on the optimizer.
3. Listens to the progress of the solve and displays it in the **Optimizing in Progress** dialog box.
4. Stops the optimizer during the solve process if the user presses the **Stop** button in the **Optimizing in Progress** dialog box.

Launch and monitor the solve process

The PPO GUI launches the solve process from the engine optimizer by invoking the method `IloMSEngineOptimizer.solve`. This method takes only one parameter, which is the `model` to solve. It returns a Boolean which indicates if the model is solved or not.

```
public abstract boolean solve(IloMSModel model)
```

To perform preprocessing the following method is called before the solving part:

```
public void beforeSolve()
```

To perform post processing the following method is called after the solving part:

```
public void afterSolve()
```

There are two approaches for implementing these methods:

- ♦ use the default engine optimizer
- ♦ redefine the engine optimizer.

Use the default engine optimizer

You can use the PPO default engine optimizer in your implementation if so desired.

The method `IloMSOptimizerFactory.setFactory` sets the default optimizer factory that the PPO GUI uses for solving model data. The parameter `defaultFactory` is the default optimizer factory.

```
public void setFactory(IloMSOptimizerFactory defaultFactory)
```

The following method retrieves the default factory:

```
public IloMSOptimizerFactory getFactory()
```

The class `IloMSEngineOptimizer` comes with one utility method that allows you to launch a solve on a given engine optimizer. This method can be used for launching an engine optimizer created by the default engine optimizer factory. This method ensure that first, intermediate, and last solutions found by the given optimizer will be monitored on the Optimizing in Progress dialog box.

```
public boolean solve(ilog.plant.IloMSModel model)
```

Create your own engine optimizer

It is possible to redefine an engine optimizer implementation to be used in PPO instead of the default. To do that, you need to define a new class inheriting from `IloMSEngineOptimizer`. In this new class, you need to specify and implement the solve method. The method `IloMSOptimizerFactory.setFactory` sets the default optimizer factory that the PPO GUI uses for solving model data. The parameter `factory` is the default optimizer factory.

```
public void setFactory(IloMSOptimizerFactory defaultFactory)
```

The following method retrieves the default factory:

```
public IloMSOptimizerFactory getFactory()
```

The class `IloMSEngineOptimizer` comes with one utility method that allows you to launch a solve on a given engine optimizer. This method can be used for launching an engine optimizer created by the default engine optimizer factory. This method ensures that first, intermediate, and last solutions found by the given optimizer will be monitored on the Optimizing in Progress dialog box.

```
public boolean solve(ilog.plant.IloMSModel model)
```

You can also extend the optimizer by writing preprocessing and post processing methods called (respectively) before and after the solve method.

```
public void beforeSolve()
public void afterSolve()
```

Stopping the solve process

While a solve is in progress in PPO, it is possible to interrupt the process by pressing the **Stop** button on the **Optimizing in Progress** dialog box. The engine optimizer will be notified of this event when the method `stop` is called:

```
public void stop()
```

By default, this method invokes the `stop()` method on all engine optimizers currently processing a solve launched with the method `solve(ilog.plant.IloMSModel)`.

Configuring the data views

The data views include the **Master Data** and **Transactional Data** views which consist of tables that contain the model data and generated plan data. You can add, remove or modify data tables per your needs using the techniques of this section.

In this section

Overview
A look at the default version of the data tables.

Activate the customization in a plug-in
First the customization must be activated.

Add a new table
Uses an example to describe how to add a new data table to your model as displayed in PPO.

Remove a table
Some data tables may not be useful in your model.

Modify a table
You can modify various aspects of a table in a data view.

Overview

In this section you will learn how to:

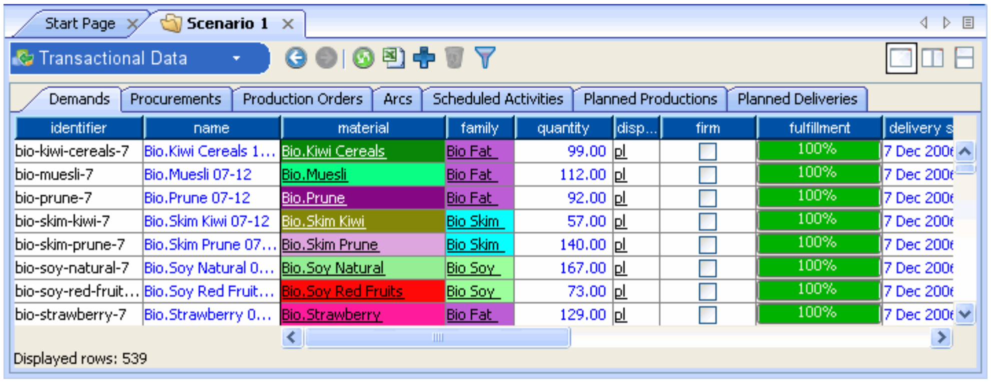
♦ *Activate the customization in a plug-in*

♦ *Add a new table*

♦ *Remove a table*

♦ *Modify a table*

By default the **Transactional Data** view has an appearance similar to the following image. The content of this view is defined by two `.xml` files: `data/gui/table/datamodeldisplay.xml` and `data/gui/table/datamodelaccess.xml`. Also note the corresponding files that control the content of the **Master Data** views: `data/gui/table/masterdatadisplay.xml` and `data/gui/table/masterdataaccess.xml`.



identifier	name	material	family	quantity	disp...	firm	fulfillment	delivery s
bio-kiwi-cereals-7	Bio.Kiwi Cereals 1...	Bio.Kiwi Cereals	Bio Fat	99.00	pl	☐	100%	7 Dec 200
bio-muesli-7	Bio.Muesli 07-12	Bio.Muesli	Bio Fat	112.00	pl	☐	100%	7 Dec 200
bio-prune-7	Bio.Prune 07-12	Bio.Prune	Bio Fat	92.00	pl	☐	100%	7 Dec 200
bio-skim-kiwi-7	Bio.Skim Kiwi 07-12	Bio.Skim Kiwi	Bio Skim	57.00	pl	☐	100%	7 Dec 200
bio-skim-prune-7	Bio.Skim Prune 07...	Bio.Skim Prune	Bio Skim	140.00	pl	☐	100%	7 Dec 200
bio-soy-natural-7	Bio.Soy Natural 0...	Bio.Soy Natural	Bio Soy	167.00	pl	☐	100%	7 Dec 200
bio-soy-red-fruit...	Bio.Soy Red Fruit...	Bio.Soy Red Fruits	Bio Soy	73.00	pl	☐	100%	7 Dec 200
bio-strawberry-7	Bio.Strawberry 0...	Bio.Strawberry	Bio Fat	129.00	pl	☐	100%	7 Dec 200

Displayed rows: 539

For various reasons you may want to customize the content of the data views. In order to not directly modify the PPO `.xml` files listed earlier, an optional “dataaccess” file and an optional “display” file can be specified to redefine the content of this view by adding, removing, or modifying tables. In the following sections, these optional files will be referred to as `dataaccess.xml` and `display.xml`.

Activate the customization in a plug-in

The class `ilog.plant.gui.docview.IloMSMasterDataPanel` is the implementation of the master data view. A static method needs to be called to specify the names and location of the optional `dataaccess.xml` and `display.xml`.

```
public static void SetOptionalDataFileName(String directory,
                                           String dataAccessFileName,
                                               String displayFileName)
```

If the specified files cannot be found, an error is raised in the console. The default value for the file names is “null” which means that there is no customization of the view.

The `SetOptionalDataFileName` method needs to be called before any creation of a Data view. One solution is to call it within the class `Plugin`:

```
public class PluginInstaller extends Plugin
{
  String DIRECTORY = "../plugins/myplugin/data/gui/table/";
  String ACCESS_FILE = "mydataaccess.xml";
  String DISPLAY_FILE = "mydisplay.xml";

  public void install(AbstractApplication absApp)
  {
    IloMSMasterDataPanel.SetOptionalDataFileName(DIRECTORY, ACCESS_FILE,
DISPLAY_FILE);
    …
```

Add a new table

Uses an example to describe how to add a new data table to your model as displayed in PPO.

In this section

Overview
An overview of the section.

Define the JavacallsView
First we need to gather the appropriate data.

Write the data access
The data is then organized.

Write the display
The data is then written to the new table.

Overview

In this example a new custom **Shipments** table is added to the Data view; this will allow the GUI user to see and edit the existing `CustomShipment` objects.

Define the JavacallsView

First the `JavacallsView` needs to be extended to return an iterator over all the `CustomShipment` objects. Since we are not interested in adding or removing a `CustomShipment`, the methods `create`, `add` and `delete` are not used.

```
/**
 * Date provider for the <code>CustomShipment</code>
 */
public class ShipmentDataProvider implements JavacallsView
{
  /**
   * The data object passed as parameter for the creation of the view
   */
  protected CommonTableData data;

  public void setDataModel(Object dataModel)
  {
    data = (CommonTableData) dataModel;
  }

  /**
   * Informs if a cell is readonly or not
   * @return true if is readonly
   */
  public boolean isReadOnly(Object ref, String identifier)
  {
    return false;
  }

  /**
   * Returns an iterator over a list of <code>CustomShipment</code>.
   */
  public Iterator iterator()
  {
    Collection rowList = new ArrayList();
    // . . . . retrieve instances
    return rowList.iterator();
  }

  public Object create(Object[] parameters) {
    return null;
  }

  public boolean add(Object row) {
    return false;
  }

  public boolean delete(Object row) {
    return false;
  }
  public boolean save() {
    return true;
  }
}
```

Write the data access

The `masterdataaccess.xml` file defines the getter and setter for all the columns of the **Master Data** view, and `datamodelaccess.xml` for the **Transactional Data** view. The column `CategoryCode` is retrieved from a custom property.

```
<DatabaseEditor>
  <Views>
    <View id="ShipmentView" type="java"
provider="mypackage.ShipmentDataProvider"
          rowtype="CustomShipment">
      <Columns>
        <Column id="Identifier" type="java.lang.String"
neededforcreation="true">
          <Get call="getIdentifier"/>
          <Set call="setIdentifier"/>
        </Column>
        <Column id="Name" type="java.lang.String" neededforcreation="true">
          <Get call="getName"/>
          <Set call="setName"/>
        </Column>
        <Column id="CategoryCode" type="java.lang.String">
          <Get call="getStringProperty">
            <Parameters>
              <Parameter type="java.lang.String" value="CATEGORYCODE"/>
            </Parameters>
          </Get>
      </Columns>
    </View>
  </Views>
</DatabaseEditor>
```

Write the display

Write the `display.xml` file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DatabaseEditor>
  <Views>
    <View id="ShipmentView" label="Shipments" forcereadyonly="true">
      <Columns>
        <Column id="Name" label="Tables.name" width="100" freeze="true"
                      readonly="true"/>
        <Column id="CategoryCode" label="category code" width="72"/>
      </Columns>
    </View>
  </Views>
</DatabaseEditor>
```

Now the new table should appear at the end of the existing tables.

At this time it is not possible to specify the position of the optional tables; they are added at the end of the standard PPO tables.

Remove a table

It is possible to remove an existing table simply by redefining the table to have no columns.

To do this, the `dataaccess.xml` file and the `display.xml` file need to contain a redefinition for the table without any columns; it is then presumed that the table is not displayable.

In the example below, no columns are defined for the **Recipes** table, so it will not display in the GUI.

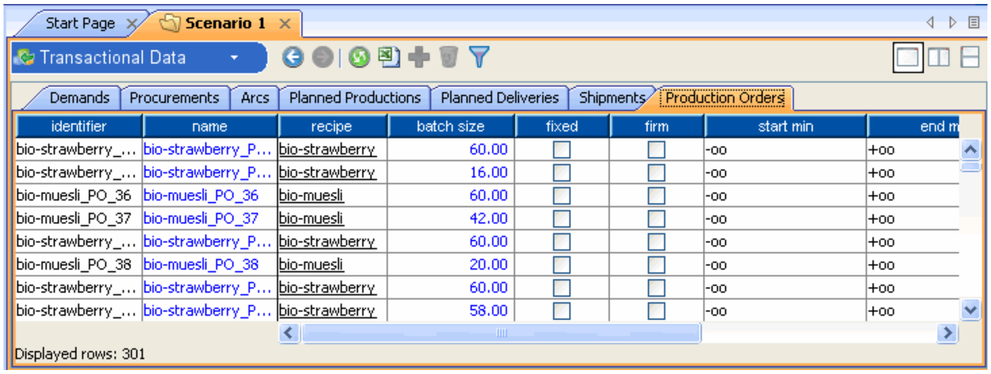
```
    <View id="RecipeView" label="Tables.Recipes">
      <Columns>
      </Columns>
    </View>

    <View id="RecipeView" type="java"
provider="ilog.plant.gui.model.table.data.IloMSRecipeDataProvider"
          rowtype="ilog.plant.IloMSRecipe">
      <Columns>
       </Columns>
    </View>
```

Modify a table

In order to redefine an existing table, the optional `dataaccess.xml` and `display.xml` files need to contain a new definition of the table; that is, a table with the same “view ID".

This new definition will replace the PPO standard version, and will then be added at the end of the existing PPO tables.

As an example, here is what the **Transactional Data** view looks like after adding the **Shipments** table, removing the **Scheduled Activities** table, and modifying the **Production Orders** table (it appears last).



identifier	name	recipe	batch size	fixed	firm	start min	end m
bio-strawberry_...	bio-strawberry_P...	bio-strawberry	60.00	☐	☐	-oo	+oo
bio-strawberry_...	bio-strawberry_P...	bio-strawberry	16.00	☐	☐	-oo	+oo
bio-muesli_PO_36	bio-muesli_PO_36	bio-muesli	60.00	☐	☐	-oo	+oo
bio-muesli_PO_37	bio-muesli_PO_37	bio-muesli	42.00	☐	☐	-oo	+oo
bio-strawberry_...	bio-strawberry_P...	bio-strawberry	60.00	☐	☐	-oo	+oo
bio-muesli_PO_38	bio-muesli_PO_38	bio-muesli	20.00	☐	☐	-oo	+oo
bio-strawberry_...	bio-strawberry_P...	bio-strawberry	60.00	☐	☐	-oo	+oo
bio-strawberry_...	bio-strawberry_P...	bio-strawberry	58.00	☐	☐	-oo	+oo

Displayed rows: 301

Database customization

This describes customizations that you can make regarding database usage in PPO. The list of supported databases is available at *Database usage and connectivity*.

In this section

Installing JDBC jars
This sections details how to install JDBC jars for database servers.

Customizing PPO data model tables
You can customize the default PPO data model by adding or removing columns and tables.

Installing JDBC jars

This sections details how to install JDBC jars for database servers.

In this section

Oracle
Setting up the client.

Microsoft SQL Server 2000 and 2005
Setting up the client.

Oracle

PPO uses the JDBC Thin driver. You should download the Oracle® client from Oracle web site, install it on your machine and copy the file `ojdbc14.jar` from the Oracle client installation directory to the PPO library directory at `<PPO_INSTALL_DIR>/lib/jar`. We recommend downloading the Oracle Database 10g or 11g client. You must restart PPO in order to load the new class path.

Microsoft SQL Server 2000 and 2005

To use SQL Server® with PPO you have to download the latest version of Microsoft® JDBC driver (the JDBC driver 2005) and install it on your machine. Then you should copy the `sqljdbc.jar` file from the JDBC driver installation directory to the PPO library directory at `<PPO_INSTALL_DIR>/lib/jar`. You must restart PPO in order to load the new class path.

Customizing PPO data model tables

You can customize the default PPO data model by adding or removing columns and tables.

In this section

PPO persistence customization files
Proceed with caution when editing files in the resources directory.

Removing tables and columns from the PPO model
Methods to remove tables and columns.

Adding property columns to the PPO model
Methods of adding columns as a property to a PPO object.

Adding custom tables
Describes how to add custom tables to the data model.

Choosing between using properties or custom tables
Factors to help you decide.

PPO persistence customization files

The directory `<PPO_INSTALL_DIR>/data/database/resources` contains a set of files used by PPO to read and write data to a database. Please be careful when changing the content of these files.

♦ database.xml: This file contains the PPO database schema in *Turbine* XML format used by Apache DdlUtils (see *http://db.apache.org/ddlutils* for a detailed description of the syntax). This file is used when creating the database from PPO (or when exporting the `.ddl` file). This file is validated by the XML reader using `database.dtd` file.

Warning: Do not change or remove the `database.dtd` file.

♦ repository.xml: This file contains the PPO object-relational mapping. It describes the mapping between the object-oriented model of PPO and the relational database persistence. PPO uses the Apache OJB as a mapping layer (see *http://db.apache.org/ojb/* for more details). This file *includes* at runtime.

♦ repository_database.xml, repository_internal.xml and repository_PPO.xml. All these files are validated against the `repository.dtd` file. The OO-relational mapping of the PPO model is contained in `repository_PPO.xml`.

♦ PPOSchema.mdb: This file contains the PPO database schema in Microsoft® Office Access® format. This file is used when saving a scenario in Access format.

♦ CustomTables.xml: This contains an example of OO-relational mapping of custom tables added to PPO model (see below).

♦ PPOCustomSchema.mdb: This contains an example of a customized PPO schema with a custom table added to the model.

♦ persistence.properties: This is a configuration file of PPO persistence and mapping layer (see below).

♦ Log4j.properties: This is the configuration file of logging/tracing functionalities of the PPO persistence and mapping layer (see *http://logging.apache.org/log4j/* for more information).

Warning: The following files should never be changed or modified: `OJB.properties`, `repository.dtd`, `database.dtd` and `repository_internal.xml`.

Removing tables and columns from the PPO model

The PPO data model covers a wide range of plant modeling. For some applications, some of the available tables or columns are not relevant. PPO allows you to simplify the model and keep only the necessary entities used by your application.

Oracle and Microsoft SQLServer

To remove unnecessary tables or columns you need to edit `database.xml` and `repository_PPO.xml` files.

You should remove all elements related to the tables and columns you want to remove from these files.

Microsoft Access

Start by copying and renaming the file `PPOSchema.mdb`. Edit the copied file and remove the unnecessary tables and columns. Edit `persistence.properties` and change the value of the property `Persistence.MSMdbSchemaFile`. Set it to the name of your customized `.mdb` file. You can keep your customized `.mdb` file in the same directory as the original one.

Adding property columns to the PPO model

As with `.csv` files, PPO database files can be customized by adding columns as a property to the corresponding PPO objects (instances of the `IloMSObject` class).

A property column must have a name of the format: `PROPERTY_XXX_NAME`.

`XXX` is the type of column and `NAME` is the property name used when accessing the property value in memory (using the methods `IloMSObject.getXXXProperty()` and `IloMSObject.setXXXProperty()`).

`XXX` may have one of the following values:

- ♦ `STRING` for string values
- ♦ `NUM` for float/double values
- ♦ `INT` for integer values.

Oracle and Microsoft SQL Server

To add properties to an Oracle® or Microsoft® SQL Server® database you must:

Modify the file `database.xml` to add the property columns (this file is used to create the database).

Then modify the file `repository_PPO.xml` to add the mapping of the property columns as follows:

```
<field-descriptor name="PROPERTY_INT_PROPERTYNAME"
    column="PROPERTY_INT_PROPERTYNAME"
    jdbc-type="INTEGER">
</field-descriptor>
```

The field name should be identical to the column name and start with PROPERTY_XXX. The JDBC-type must correspond to the type in the column name. The JDBC-type should be `INTEGER` for `INT` properties, `FLOAT` for `NUM` properties and `VARCHAR` for `STRING` properties.

Microsoft Access

The PPO reader and writer for Microsoft Access® handles properties without any customization. When you load an `.mdb` file with property columns, property values will be assigned to the corresponding objects. Symmetrically, when you save a PPO model with properties, the corresponding columns are created automatically in Access tables.

Adding custom tables

PPO allows adding custom tables to the data model, and offers a simple way to load and save custom tables.

To add custom tables:

♦ Define the structure of the tables in xml format

♦ Create a model extension point in a plug-in

♦ Run the PPO persistence generator script to generate all necessary Java™ code to handle the persistence of the tables

♦ Compile and deploy your plug-in with the custom tables

Once the plug-in is deployed, PPO will handle the load and the save of your custom tables in Microsoft® Access® , database servers (Oracle® and SQL Server®) and Excel® format. PPO will also generate ddl code for your tables.

Custom tables declaration

The structure of custom tables must be defined in an xml format as follows:

```
<datamodel>
  <doc>
    Custom data model example.
  </doc>
  <tables>
    <table name="BUCKET_DATE" topic="custom">
      <field name="BUCKET_ID" type="id" key="true">
      </field>
      <field name="BUCKET_SEQUENCE_ID" type="id" key="false">
      </field>
      <field name="START_DATE" type="date" key="false">
      </field>
      <field name="END_DATE" type="date" key="false">
      </field>
    </table>
    <table name="HARVEST" topic="custom">
      <field name="HARVEST_ID" type="id" key="true">
      </field>
      <field name="MATERIAL_ID" type="id" key="true">
      </field>
      <field name="DAY" type="integer" key="true">
      </field>
      <field name="QUANTITY" type="double" key="false">
      <default>0</default>
      </field>
     </table>
   </tables>
```

```
</datamodel>
```

A table is a set of fields and can have a *topic* attribute. Topics are used to categorize tables when exporting a scenario to Excel. A field must have a type attribute. The type can have one of the following values:

- `id` for identifiers
- `string` for strings
- `double` for floating-point values
- `int` for integers
- `date` for date values

If the field is a primary key, the attribute `key` must be set to true. A field is mandatory unless it has a default set by the `<default>` tag.

Creating the model extension point in the plug-in

There is an example plug-in file in the <*PPOInstallDirectory*>`/examples/customTables` directory. In the `plugin.xml` you must add the xml element defining the extension point as follows:

```
<extension point="ModelExtension" id="Custom" name="Custom"
          generationPackage="ilog.plant.examples.customTables.tables"
          classname="ilog.plant.examples.customTables.CustomModelExtension"
          loadBefore="true">
</extension>
```

The `id` attribute of the element is used to prefix some of the generated files. The `generationPackage` defines the name of the Java package where the code will be generated. The `classname` provides the Java class name of the model extension. This class must extend `ilog.plant.persistence.table.ModelExtension`. Your model extension class will be used to access the custom tables when loading from or saving to the database.

Generating custom tables Java code

In the <*PPOInstallDirectory*>`/examples/customTables` directory there is a `build.xml` file that you can use to call PPO script to generate Java code for the persistence layer of your custom tables. Edit the `customTables.properties` file to set the variable `ModelExtension.ModelDefintionFile` to the xml file containing the custom table definitions. Then run `ant build`. The Java code is generated, compiled and deployed with your plug-in.

Choosing between using properties or custom tables

Extending the data model with properties is equivalent to adding new attributes to PPO objects. You should use properties when the data you want to add is directly related to PPO objects (one-to-one mapping) and the number of added properties must be only a few (two or three properties per table).

When the mapping between the custom data and PPO objects is not one-to-one, or when the amount of custom data is significant, we highly recommend the use of custom tables for readability and performance.

Options of the PlantPowerOps executable file

The file `PlantPowerOps.lax` located in <*PPOInstallDirectory*>`\bin` is a property file associated with `PlantPowerOps.exe`. By modifying this file you can direct PPO output to a DOS window and decrease the memory allocated to Java™ processes.

Displaying the DOS console

Open the file `PlantPowerOps.lax` in an editor and do the following:

- Search for `lax.stderr.redirect` and change it to: `lax.stderr.redirect=console`.
- Search for `lax.stdout.redirect` and change it to: `lax.stdout.redirect=console`.
- Save and close the file `PlantPowerOps.lax`.
- Launch PPO as usual, and the output will be redirected to a DOS window.

Decreasing memory allocated to Java

If you encounter big memory allocation problems, you may want to decrease the amount allocated to the GUI; this increases the memory available for optimization processes.

Open the file `PlantPowerOps.lax` in an editor and do the following:

- Search for `lax.nl.java.option.additional`. It may appear as follows:

  ```
  lax.nl.java.option.additional=-Xmx512m -Djava.library.path=..\\lib\\x86_.
  net2003_7.1\\dll_mda;..\\lib\\x86_.net2005_8.0\\dll_mda;. -Duser.language=en
  -Duser.country=US
  ```

- To decrease the memory allocated to the GUI, in the previous line change `-Xmx512m` to `-Xmx256m`.

Using PPO with Microsoft products

This section describes how to build and run examples delivered with Plant PowerOps when you use Microsoft® Visual C++ .NET and Microsoft Windows® XP. Included are instructions on creating a project and linking the target with PPO.

In this section

Overview
The PPO dynamic link library.

Build and run Plant PowerOps examples
How to build and run the examples.

Creating a project workspace and link the target with PPO
This section describes how to create a project workspace and linked target.

Overview

Throughout this section, `<PPO>` refers to the directory in which Plant PowerOps is installed. For instance, if PPO is installed in the default directory `C:\ILOG\PowerOps\Plantxx`, then `<PPO>\include` refers to `C:\ILOG\PowerOps\Plantxx\include`.

The PPO dynamic link library is delivered in multi-threaded-DLL format with a static library. This format uses the new standard template library (STL) and is compiled using the namespace std.

The PPO dynamic link library and the corresponding static library are in the directory:

```
<PPO>\lib\x86_.net2005_8.0\dll_mda\plant32.dll
<PPO>\lib\x86_.net2005_8.0\dll_mda\plant32.lib
```

The following information refers to the multi-threaded-DLL STL dynamic link library. Please refer to the *Code Generation* section of the MSVC++ .NET online manual for more information.

As you will be using the STL, you should define the macro `IL_STD` before compiling.

Build and run Plant PowerOps examples

The Plant PowerOps examples have all been gathered into one project. As explained above, the following information applies to the multi-threaded-DLL STL dynamic link library. The related file is:

`<PPO>\examples\x86_.net2005_8.0\dll_mda\examples.sln`

Note that the order of instructions is important.

1. Start Microsoft® Visual Studio .NET.
2. From the **File** menu, choose **Open Solution**. The **Open Solution** dialog box appears. The default selection in the **List Files Of Type** drop-down list is **Solution Files (*.SLN)**. Select the drive and set the directory to:`<PPO>\examples\x86_.net2005_8.0\dll_mda`
3. Select the **examples.sln** file and click **Open**. To list all the projects in the solution, choose **Solution Explorer** from the **View** menu.
4. To build only one example:
   a. Select the **api02exercise1** project in the **Solution Explorer** window.
   b. From the **Build** menu, choose **Build api02exercise1**. Wait for the building process to complete.
   c. Start a command prompt, and in the window set the path to the PPO dynamic link library if it has not already set. Type:`set PATH=%PATH%;<PPO>\lib\x86_.net2005_8.0\dll_mda`
   d. Next, at the command prompt enter the following (where `<dataExample>` is the name of a data example from `<PPO>\data\*.csv`):

   ```
   <PPO>\examples\x86_.net2005_8.0\dll_mda\api02exercise1 -instance <PPO>\
   examples\data\<dataExample>
   ```

   The result is then displayed.
5. To build all the examples:
   a. From the **Build** menu, choose **Build Solution**.
   b. Start a command prompt, and in the window set the path to the PPO dynamic link library if it has not already set. Type:`set PATH=%PATH%;<PPO>\lib\x86_.net2005_8.0\dll_mda`
   c. Next, at the command prompt enter the following (where `<name>` is the example to execute, and `<dataExample>` is the name of the data example from `<PPO>\examples\data\*.csv`):`<PPO>\examples\x86_.net2005_8.0\dll_mda\<name> <PPO>\examples\data\<dataExample>`
   d. You can also type (where `600` is the maximum time limit for execution):

   ```
   <PPO>\examples\x86_.net2005_8.0\dll_mda\<name> -instance <PPO>\examples\
   data\<dataExample> -runtime 600
   ```

   The result is then displayed.

Creating a project workspace and link the target with PPO

This procedure assumes:

- ♦ A source file named `test.cpp` that uses the API of the PPO library;
- ♦ A directory `<MYAPPDIR>` in which this file is located;
- ♦ A target named `test.exe`.
- ♦ A first step of building a `test.sln` solution.

Note that the order of instructions in this procedure is important.

1. Start Microsoft® Visual Studio .NET.
2. Start the Win32 Application Wizard:
   - a. From the **File** menu, select **New-> Blank Solution**. The **New Project** dialog box appears.
   - b. In the **Project Types** pane, select **Visual C++ Projects**.
   - c. In the **Templates** pane, select the **Win32 Project** icon.
   - d. Fill in the project name (**test**). If necessary, correct the location of the project (referred to hereafter as `<MYAPPDIR>`).
   - e. Click **OK**.

   The Win32 Application Wizard appears.
3. Check application settings, display the Solution Explorer, and open `test.cpp`:
   - a. Click **Application Settings**, and select **Console Application** as the application type.
   - b. Make sure that **Empty project** is checked in **Additional Options**, and then click **Finish**.
   - c. Select **Solution Explorer** from the **View** menu (if not already displayed).
   - d. From the **Project** menu, choose **Add Existing Item**.
   - e. Select **test.cpp**, and click **Open**.
4. Next, set options so that the project finds the Plant PowerOps library and include files:
   - a. From the **Project** menu, choose **Properties**. The **Project Properties** dialog box appears.
   - b. In the **Configuration** drop-down list, select **Release**.
   - c. In the left pane, click **Configuration Properties**, then **C/C++**.
   - d. In the **General Category** tab:
     - ♦ In the **additional include directories** text field, add the directory **<ppo>\include**.

- ♦ For **Debug Information Format**, choose **Line Numbers Only (/Zd)**.
- ♦ Choose **No** for **Detect 64-bit Portability Issues**.

   e. In the **Preprocessor** pane, add **IL_STD** to the **Preprocessor Definitions** text field.

   f. In the **Code Generation** tab, set **Runtime library** to **Multi-threaded DLL (/MD)**.

5. Continuing:

   a. In the **Configuration Properties** tree, choose **Linker**.

   b. In the **Input** pane, add two files: **wsock32.lib** and **<ppo>\lib\x86_.net2005_8.0\dll_mda\plant32.lib**.

   c. In the **Debug** pane, set **Generate Debug Info** to **Yes (/DEBUG)**.

   d. In the **Optimization** tab, make sure that **References** and **Enable COMDAT Folding** options are set to **Default**. Then select **OK**,

6. Set **test Win32 Release** as the default project configuration and build the project:

   a. From the **Build** menu, select **Configuration Manager**.

   b. Select **Release** in the **Active Solution Configuration** drop-down list. Click **Close**.

   c. From the **Build** menu, select **Build Solution**.

7. After completion of the compiling and linking process, the target is created. Next, start a command prompt, and in the window set the path to the PPO dynamic link library if it has not already set. Type:

```
set PATH=%PATH%;<PPO>\lib\x86_.net2005_8.0\dll_mda
```

8. Run your example. The full path of `test.exe` is `<MYAPPDIR>\Release\test.exe`.

Important note

From the point of view of using Plant PowerOps, the only differences between the Win32 Release and Win32 Debug targets are that the `NDEBUG` macro is defined for the *Release* configuration, and that the `NDEBUG` macro is not defined for the *Debug* configuration.

This is why we have suggested using **Release** in the `test.sln` example, instead of the default proposed by Visual C++ .NET. Refer to the *Microsoft Visual C++ Reference Manual* for complete information on release and debug configurations.

Entity Relationship Diagrams

Presents the available Entity Relationship Diagrams (ERDs).

In this section

General tables
Shows general PPO entities like model, profiles, and criterion weights.

Activities, modes, and recipes
Shows PPO model of recipes, prototype activities, setup activities and resource modes.

Cleanup constraints
Shows entities of cleanup recipes.

Demands
Shows relations between demands, materials and storage units.

Manufacturing resources
Shows the entities that affect the availability, use, and cost of resources.

Material flow
Shows entities involved in modeling material flows related to production orders and activities.

Materials and storage units
Shows relations between materials, storage units and inventory cost functions.

Procurements
Shows procurement model in PPO.

Production orders
Shows entities involved in modeling material flows related to production orders and activities.

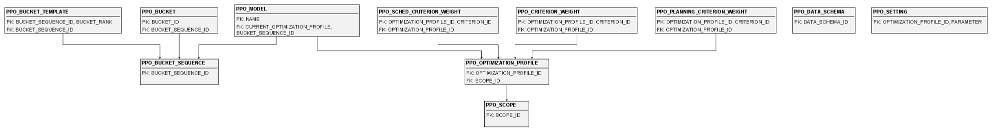
Production plans
Shows relations between demand and planned production entities.

Production schedules
Shows relations between production orders and scheduled activities.

Setup times and setup costs
Shows setup matrices and setup states in PPO.

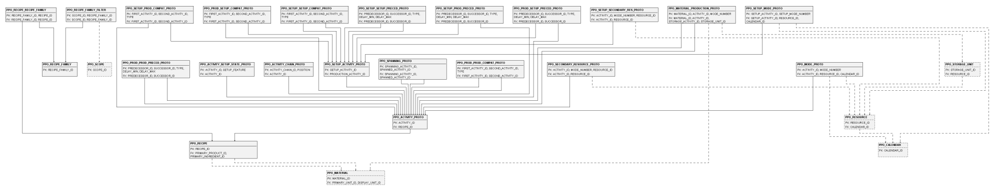
General tables

Shows general PPO entities like profile, model and criterion weights.


PPO_BUCKET_TEMPLATE
PK: BUCKET_SEQUENCE_ID, BUCKET_RANK
FK: BUCKET_SEQUENCE_ID
PPO_BUCKET
PK: BUCKET_ID
FK: BUCKET_SEQUENCE_ID
PPO_MODEL
PK: NAME
FK: CURRENT_OPTIMIZATION_PROFILE, BUCKET_SEQUENCE_ID
PPO_SCHED_CRITERION_WEIGHT
PK: OPTIMIZATION_PROFILE_ID, CRITERION_ID
FK: OPTIMIZATION_PROFILE_ID
PPO_CRITERION_WEIGHT
PK: OPTIMIZATION_PROFILE_ID, CRITERION_ID
FK: OPTIMIZATION_PROFILE_ID
PPO_PLANNING_CRITERION_WEIGHT
PK: OPTIMIZATION_PROFILE_ID, CRITERION_ID
FK: OPTIMIZATION_PROFILE_ID
PPO_DATA_SCHEMA
PK: DATA_SCHEMA_ID
PPO_SETTING
PK: OPTIMIZATION_PROFILE_ID, PARAMETER
PPO_BUCKET_SEQUENCE
PK: BUCKET_SEQUENCE_ID
PPO_OPTIMIZATION_PROFILE
PK: OPTIMIZATION_PROFILE_ID
FK: SCOPE_ID
PPO_SCOPE
PK: SCOPE_ID

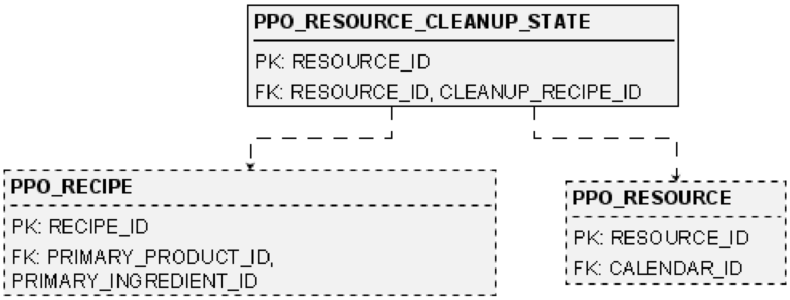
Activities, modes, and recipes

Shows PPO model of recipes, prototype activities, setup activities and resource modes.


PPO_RECIPE_RECIPE_FAMILY
PK: RECIPE_FAMILY_ID, RECIPE_ID
FK: RECIPE_FAMILY_ID, RECIPE_ID
PPO_RECIPE_FAMILY_FILTER
PK: SCOPE_ID, RECIPE_FAMILY_ID
FK: SCOPE_ID, RECIPE_FAMILY_ID
PPO_SETUP_PROD_COMPAT_PROTO
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_PROD_SETUP_COMPAT_PROTO
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_SETUP_SETUP_COMPAT_PROTO
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_SETUP_SETUP_PRECED_PROTO
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_SETUP_PROD_PRECED_PROTO
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_PROD_SETUP_PRECED_PROTO
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_SETUP_SECONDARY_RES_PROTO
PK: ACTIVITY_ID, MODE_NUMBER, RESOURCE_ID
FK: ACTIVITY_ID, RESOURCE_ID
PPO_MATERIAL_PRODUCTION_PROTO
PK: MATERIAL_ID, ACTIVITY_ID, MODE_NUMBER
FK: MATERIAL_ID, ACTIVITY_ID, STORAGE_ACTIVITY_ID, STORAGE_UNIT_ID
PPO_SETUP_MODE_PROTO
PK: SETUP_ACTIVITY_ID, SETUP_MODE_NUMBER
FK: SETUP_ACTIVITY_ID, RESOURCE_ID, CALENDAR_ID
PPO_RECIPE_FAMILY
PK: RECIPE_FAMILY_ID
PPO_SCOPE
PK: SCOPE_ID
PPO_PROD_PROD_PRECED_PROTO
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_ACTIVITY_SETUP_STATE_PROTO
PK: ACTIVITY_ID, SETUP_FEATURE
FK: ACTIVITY_ID
PPO_ACTIVITY_CHAIN_PROTO
PK: ACTIVITY_CHAIN_ID, POSITION
FK: ACTIVITY_ID
PPO_SETUP_ACTIVITY_PROTO
PK: SETUP_ACTIVITY_ID
FK: PRODUCTION_ACTIVITY_ID
PPO_SPANNING_PROTO
PK: SPANNING_ACTIVITY_ID, SPANNED_ACTIVITY_ID
FK: SPANNING_ACTIVITY_ID, SPANNED_ACTIVITY_ID
PPO_PROD_PROD_COMPAT_PROTO
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_SECONDARY_RESOURCE_PROTO
PK: ACTIVITY_ID, MODE_NUMBER, RESOURCE_ID
FK: ACTIVITY_ID, RESOURCE_ID
PPO_MODE_PROTO
PK: ACTIVITY_ID, MODE_NUMBER
FK: ACTIVITY_ID, RESOURCE_ID, CALENDAR_ID
PPO_STORAGE_UNIT
PK: STORAGE_UNIT_ID
FK: RESOURCE_ID
PPO_ACTIVITY_PROTO
PK: ACTIVITY_ID
FK: RECIPE_ID
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID
PPO_RECIPE
PK: RECIPE_ID
FK: PRIMARY_PRODUCT_ID, PRIMARY_INGREDIENT_ID
PPO_CALENDAR
PK: CALENDAR_ID
PPO_MATERIAL
PK: MATERIAL_ID
FK: PRIMARY_UNIT_ID, DISPLAY_UNIT_ID

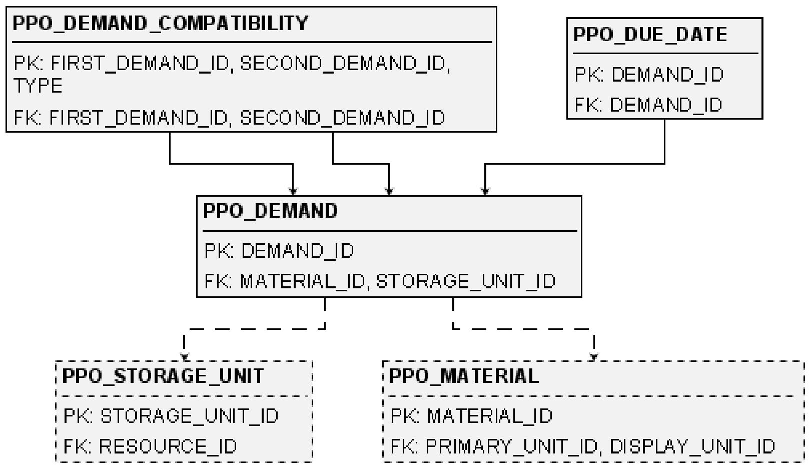
Cleanup constraints

Shows entities of cleanup recipes.


PPO_RESOURCE_CLEANUP_STATE
PK: RESOURCE_ID
FK: RESOURCE_ID, CLEANUP_RECIPE_ID
PPO_RECIPE
PK: RECIPE_ID
FK: PRIMARY_PRODUCT_ID, PRIMARY_INGREDIENT_ID
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID

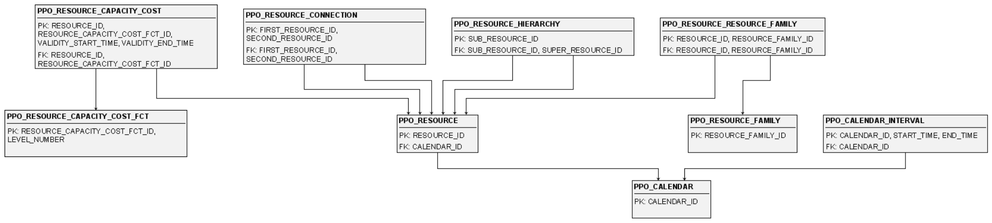
Demands

Shows relations between demands, materials and storage units.



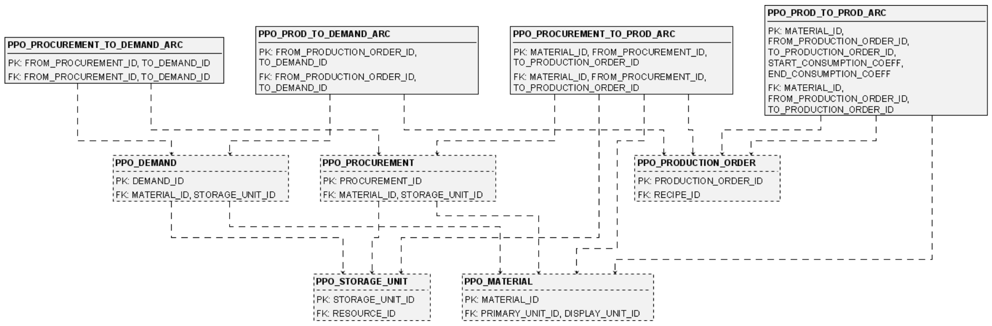
Manufacturing resources

Shows the entities that affect the availability, use, and cost of resources.


PPO_RESOURCE_CAPACITY_COST
PK: RESOURCE_ID, RESOURCE_CAPACITY_COST_FCT_ID, VALIDITY_START_TIME, VALIDITY_END_TIME
FK: RESOURCE_ID, RESOURCE_CAPACITY_COST_FCT_ID
PPO_RESOURCE_CONNECTION
PK: FIRST_RESOURCE_ID, SECOND_RESOURCE_ID
FK: FIRST_RESOURCE_ID, SECOND_RESOURCE_ID
PPO_RESOURCE_HIERARCHY
PK: SUB_RESOURCE_ID
FK: SUB_RESOURCE_ID, SUPER_RESOURCE_ID
PPO_RESOURCE_RESOURCE_FAMILY
PK: RESOURCE_ID, RESOURCE_FAMILY_ID
FK: RESOURCE_ID, RESOURCE_FAMILY_ID
PPO_RESOURCE_CAPACITY_COST_FCT
PK: RESOURCE_CAPACITY_COST_FCT_ID, LEVEL_NUMBER
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID
PPO_RESOURCE_FAMILY
PK: RESOURCE_FAMILY_ID
PPO_CALENDAR_INTERVAL
PK: CALENDAR_ID, START_TIME, END_TIME
FK: CALENDAR_ID
PPO_CALENDAR
PK: CALENDAR_ID

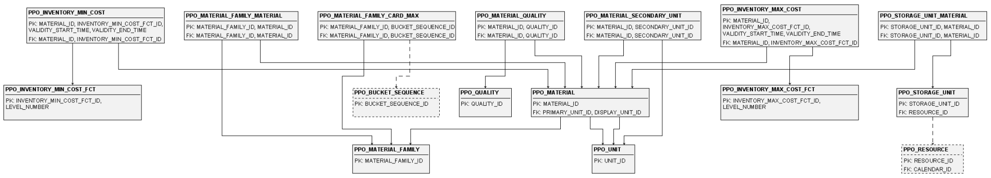
Material flow

Shows entities involved in modeling material flows related to production orders and activities.


PPO_PROCUREMENT_TO_DEMAND_ARC
PK: FROM_PROCUREMENT_ID, TO_DEMAND_ID
FK: FROM_PROCUREMENT_ID, TO_DEMAND_ID
PPO_PROD_TO_DEMAND_ARC
PK: FROM_PRODUCTION_ORDER_ID, TO_DEMAND_ID
FK: FROM_PRODUCTION_ORDER_ID, TO_DEMAND_ID
PPO_PROCUREMENT_TO_PROD_ARC
PK: MATERIAL_ID, FROM_PROCUREMENT_ID, TO_PRODUCTION_ORDER_ID
FK: MATERIAL_ID, FROM_PROCUREMENT_ID, TO_PRODUCTION_ORDER_ID
PPO_PROD_TO_PROD_ARC
PK: MATERIAL_ID, FROM_PRODUCTION_ORDER_ID, TO_PRODUCTION_ORDER_ID, START_CONSUMPTION_COEFF, END_CONSUMPTION_COEFF
FK: MATERIAL_ID, FROM_PRODUCTION_ORDER_ID, TO_PRODUCTION_ORDER_ID
PPO_DEMAND
PK: DEMAND_ID
FK: MATERIAL_ID, STORAGE_UNIT_ID
PPO_PROCUREMENT
PK: PROCUREMENT_ID
FK: MATERIAL_ID, STORAGE_UNIT_ID
PPO_PRODUCTION_ORDER
PK: PRODUCTION_ORDER_ID
FK: RECIPE_ID
PPO_STORAGE_UNIT
PK: STORAGE_UNIT_ID
FK: RESOURCE_ID
PPO_MATERIAL
PK: MATERIAL_ID
FK: PRIMARY_UNIT_ID, DISPLAY_UNIT_ID

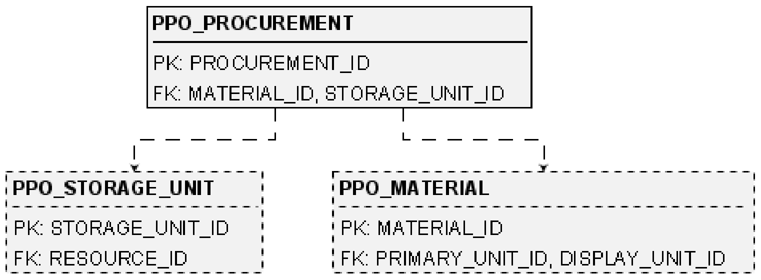
Materials and storage units

Shows relations between materials, storage units and inventory cost functions.


PPO_INVENTORY_MIN_COST
PK: MATERIAL_ID, INVENTORY_MIN_COST_FCT_ID, VALIDITY_START_TIME, VALIDITY_END_TIME
FK: MATERIAL_ID, INVENTORY_MIN_COST_FCT_ID
PPO_MATERIAL_FAMILY_MATERIAL
PK: MATERIAL_FAMILY_ID, MATERIAL_ID
FK: MATERIAL_FAMILY_ID, MATERIAL_ID
PPO_MATERIAL_FAMILY_CARD_MAX
PK: MATERIAL_FAMILY_ID, BUCKET_SEQUENCE_ID
FK: MATERIAL_FAMILY_ID, BUCKET_SEQUENCE_ID
PPO_MATERIAL_QUALITY
PK: MATERIAL_ID, QUALITY_ID
FK: MATERIAL_ID, QUALITY_ID
PPO_MATERIAL_SECONDARY_UNIT
PK: MATERIAL_ID, SECONDARY_UNIT_ID
FK: MATERIAL_ID, SECONDARY_UNIT_ID
PPO_INVENTORY_MAX_COST
PK: MATERIAL_ID, INVENTORY_MAX_COST_FCT_ID, VALIDITY_START_TIME, VALIDITY_END_TIME
FK: MATERIAL_ID, INVENTORY_MAX_COST_FCT_ID
PPO_STORAGE_UNIT_MATERIAL
PK: STORAGE_UNIT_ID, MATERIAL_ID
FK: STORAGE_UNIT_ID, MATERIAL_ID
PPO_INVENTORY_MIN_COST_FCT
PK: INVENTORY_MIN_COST_FCT_ID, LEVEL_NUMBER
PPO_BUCKET_SEQUENCE
PK: BUCKET_SEQUENCE_ID
PPO_QUALITY
PK: QUALITY_ID
PPO_MATERIAL
PK: MATERIAL_ID
FK: PRIMARY_UNIT_ID, DISPLAY_UNIT_ID
PPO_INVENTORY_MAX_COST_FCT
PK: INVENTORY_MAX_COST_FCT_ID, LEVEL_NUMBER
PPO_STORAGE_UNIT
PK: STORAGE_UNIT_ID
FK: RESOURCE_ID
PPO_MATERIAL_FAMILY
PK: MATERIAL_FAMILY_ID
PPO_UNIT
PK: UNIT_ID
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID

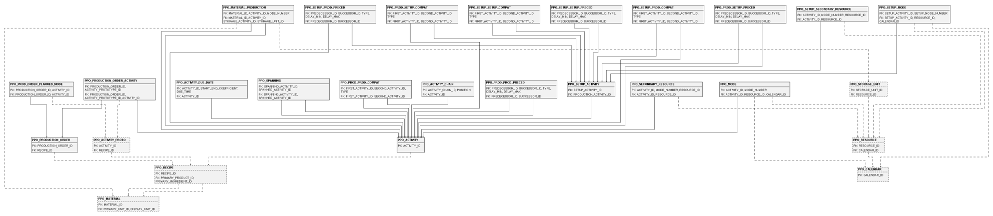
Procurements

Shows procurement model in PPO.



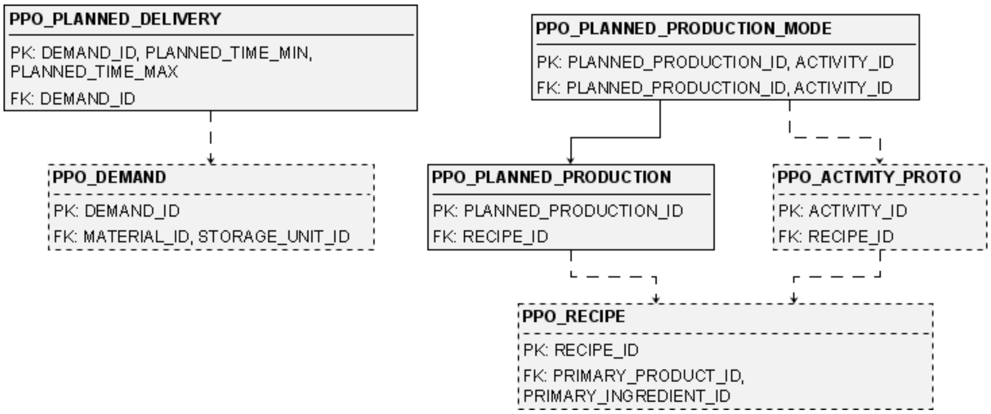
Production orders

Shows entities involved in modeling material flows related to production orders and activities.


PPO_MATERIAL_PRODUCTION
PK: MATERIAL_ID, ACTIVITY_ID, MODE_NUMBER
FK: MATERIAL_ID, ACTIVITY_ID, STORAGE_ACTIVITY_ID, STORAGE_UNIT_ID
PPO_SETUP_PROD_PRECED
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_PROD_SETUP_COMPAT
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_SETUP_SETUP_COMPAT
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_SETUP_SETUP_PRECED
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_SETUP_PROD_COMPAT
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_PROD_SETUP_PRECED
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_SETUP_SECONDARY_RESOURCE
PK: ACTIVITY_ID, MODE_NUMBER, RESOURCE_ID
FK: ACTIVITY_ID, RESOURCE_ID
PPO_SETUP_MODE
PK: SETUP_ACTIVITY_ID, SETUP_MODE_NUMBER
FK: SETUP_ACTIVITY_ID, RESOURCE_ID, CALENDAR_ID
PPO_PROD_ORDER_PLANNED_MODE
PK: PRODUCTION_ORDER_ID, ACTIVITY_ID
FK: PRODUCTION_ORDER_ID, ACTIVITY_ID
PPO_PRODUCTION_ORDER_ACTIVITY
PK: PRODUCTION_ORDER_ID, ACTIVITY_PROTOTYPE_ID
FK: PRODUCTION_ORDER_ID, ACTIVITY_PROTOTYPE_ID, ACTIVITY_ID
PPO_ACTIVITY_DUE_DATE
PK: ACTIVITY_ID, START_END_COEFFICIENT, DUE_TIME
FK: ACTIVITY_ID
PPO_SPANNING
PK: SPANNING_ACTIVITY_ID, SPANNED_ACTIVITY_ID
FK: SPANNING_ACTIVITY_ID, SPANNED_ACTIVITY_ID
PPO_PROD_PROD_COMPAT
PK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID, TYPE
FK: FIRST_ACTIVITY_ID, SECOND_ACTIVITY_ID
PPO_ACTIVITY_CHAIN
PK: ACTIVITY_CHAIN_ID, POSITION
FK: ACTIVITY_ID
PPO_PROD_PROD_PRECED
PK: PREDECESSOR_ID, SUCCESSOR_ID, TYPE, DELAY_MIN, DELAY_MAX
FK: PREDECESSOR_ID, SUCCESSOR_ID
PPO_SETUP_ACTIVITY
PK: SETUP_ACTIVITY_ID
FK: PRODUCTION_ACTIVITY_ID
PPO_SECONDARY_RESOURCE
PK: ACTIVITY_ID, MODE_NUMBER, RESOURCE_ID
FK: ACTIVITY_ID, RESOURCE_ID
PPO_MODE
PK: ACTIVITY_ID, MODE_NUMBER
FK: ACTIVITY_ID, RESOURCE_ID, CALENDAR_ID
PPO_STORAGE_UNIT
PK: STORAGE_UNIT_ID
FK: RESOURCE_ID
PPO_PRODUCTION_ORDER
PK: PRODUCTION_ORDER_ID
FK: RECIPE_ID
PPO_ACTIVITY_PROTO
PK: ACTIVITY_ID
FK: RECIPE_ID
PPO_ACTIVITY
PK: ACTIVITY_ID
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID
PPO_RECIPE
PK: RECIPE_ID
FK: PRIMARY_PRODUCT_ID, PRIMARY_INGREDIENT_ID
PPO_CALENDAR
PK: CALENDAR_ID
PPO_MATERIAL
PK: MATERIAL_ID
FK: PRIMARY_UNIT_ID, DISPLAY_UNIT_ID

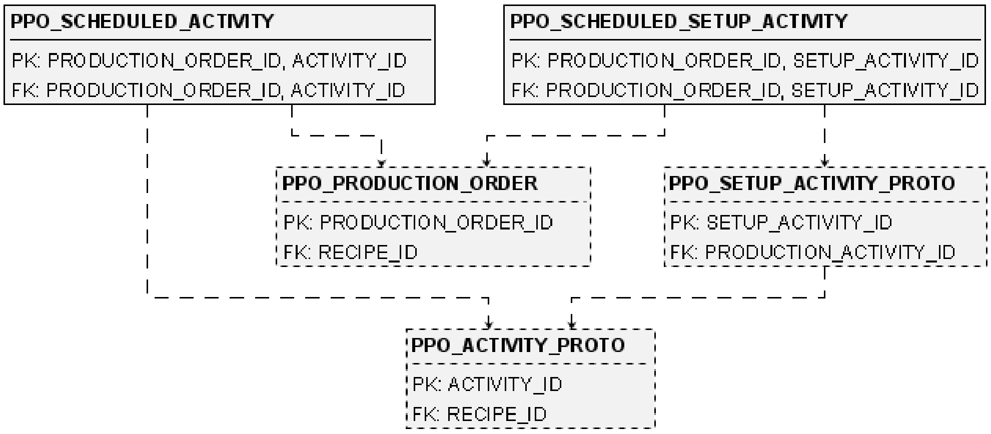
Production plans

Shows relations between demand and planned production entities.


PPO_PLANNED_DELIVERY
PK: DEMAND_ID, PLANNED_TIME_MIN, PLANNED_TIME_MAX
FK: DEMAND_ID
PPO_PLANNED_PRODUCTION_MODE
PK: PLANNED_PRODUCTION_ID, ACTIVITY_ID
FK: PLANNED_PRODUCTION_ID, ACTIVITY_ID
PPO_DEMAND
PK: DEMAND_ID
FK: MATERIAL_ID, STORAGE_UNIT_ID
PPO_PLANNED_PRODUCTION
PK: PLANNED_PRODUCTION_ID
FK: RECIPE_ID
PPO_ACTIVITY_PROTO
PK: ACTIVITY_ID
FK: RECIPE_ID
PPO_RECIPE
PK: RECIPE_ID
FK: PRIMARY_PRODUCT_ID, PRIMARY_INGREDIENT_ID

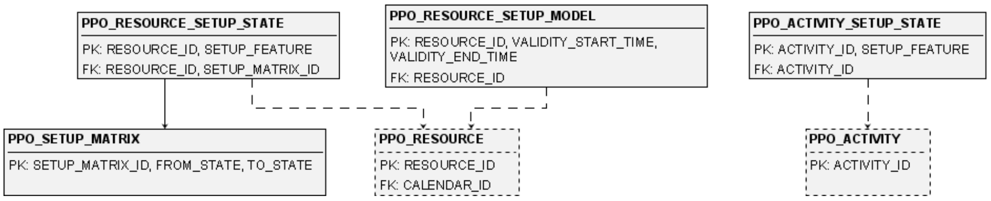
Production schedules

Shows relations between production orders and scheduled activities.


PPO_SCHEDULED_ACTIVITY
PK: PRODUCTION_ORDER_ID, ACTIVITY_ID
FK: PRODUCTION_ORDER_ID, ACTIVITY_ID
PPO_SCHEDULED_SETUP_ACTIVITY
PK: PRODUCTION_ORDER_ID, SETUP_ACTIVITY_ID
FK: PRODUCTION_ORDER_ID, SETUP_ACTIVITY_ID
PPO_PRODUCTION_ORDER
PK: PRODUCTION_ORDER_ID
FK: RECIPE_ID
PPO_SETUP_ACTIVITY_PROTO
PK: SETUP_ACTIVITY_ID
FK: PRODUCTION_ACTIVITY_ID
PPO_ACTIVITY_PROTO
PK: ACTIVITY_ID
FK: RECIPE_ID

Setup times and setup costs

Shows setup matrices and setup states in PPO.

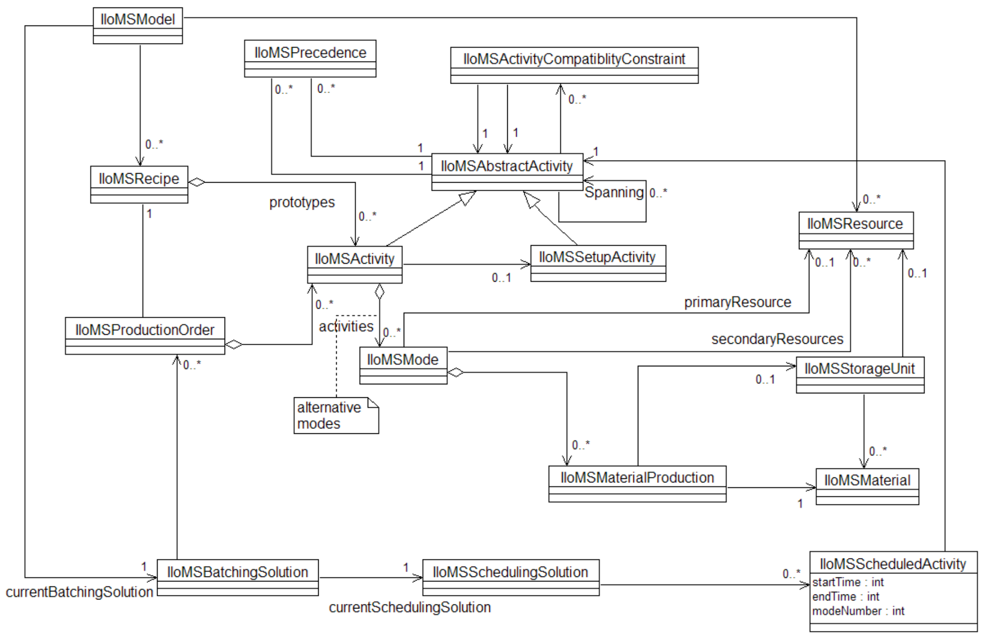

PPO_RESOURCE_SETUP_STATE
PK: RESOURCE_ID, SETUP_FEATURE
FK: RESOURCE_ID, SETUP_MATRIX_ID
PPO_RESOURCE_SETUP_MODEL
PK: RESOURCE_ID, VALIDITY_START_TIME, VALIDITY_END_TIME
FK: RESOURCE_ID
PPO_ACTIVITY_SETUP_STATE
PK: ACTIVITY_ID, SETUP_FEATURE
FK: ACTIVITY_ID
PPO_SETUP_MATRIX
PK: SETUP_MATRIX_ID, FROM_STATE, TO_STATE
PPO_RESOURCE
PK: RESOURCE_ID
FK: CALENDAR_ID
PPO_ACTIVITY
PK: ACTIVITY_ID

Universal Modeling Language diagrams

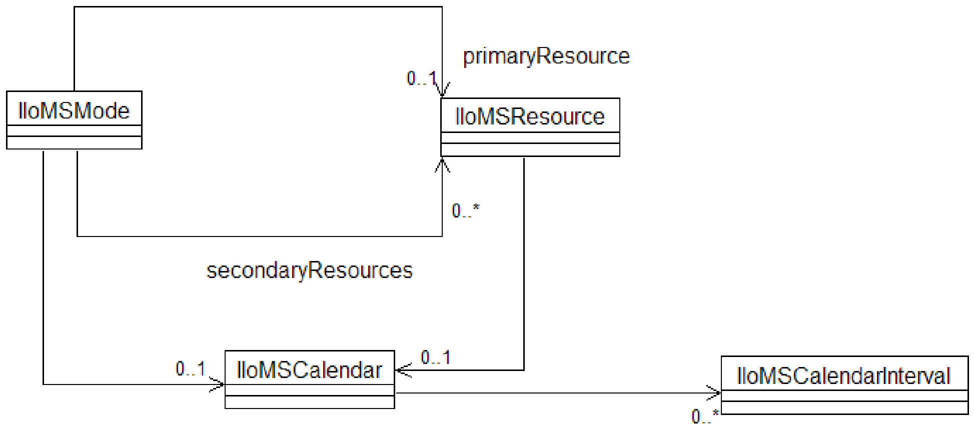
The following UML diagrams are available.

- Overview of PPO
- Calendars
- Solutions
- Pegging

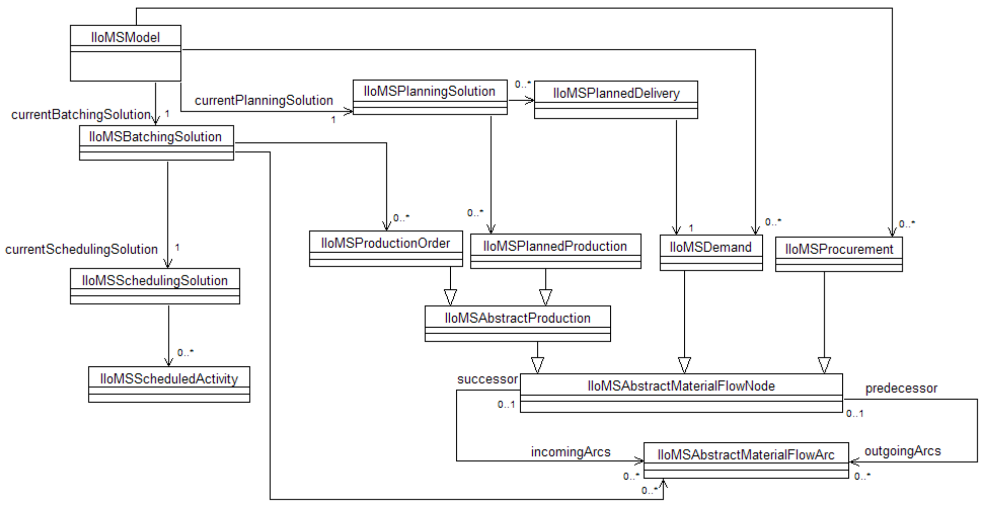
PPO UML Diagram Overview



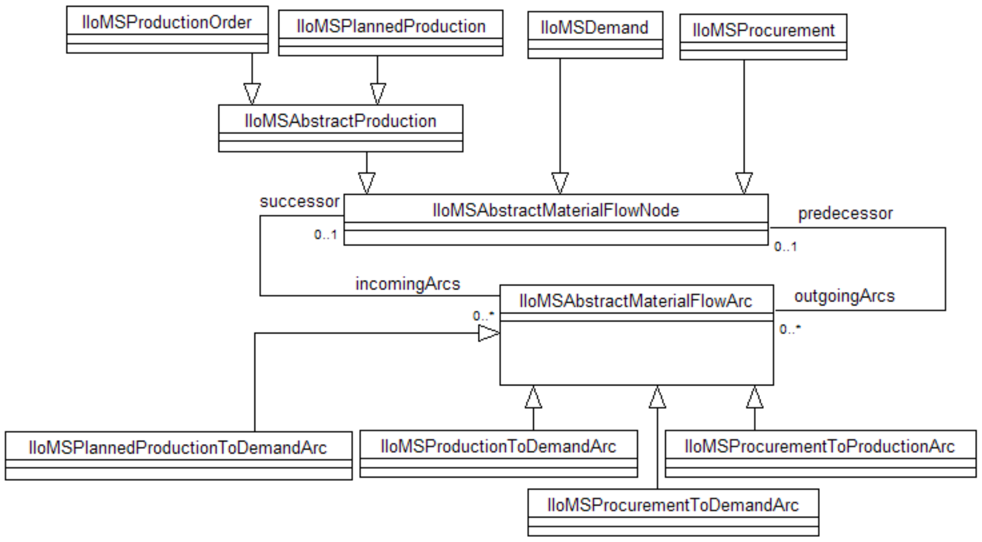
**PPO UML Diagram
Calendars**



**PPO UML Diagram
Solutions**



PPO UML Diagram
Pegging Arcs


IloMSProductionOrder
IloMSPlannedProduction
IloMSDemand
IloMSProcurement
IloMSAbstractProduction
successor
0..1
IloMSAbstractMaterialFlowNode
predecessor
0..1
incomingArcs
0..*
IloMSAbstractMaterialFlowArc
outgoingArcs
0..*
IloMSPlannedProductionToDemandArc
IloMSProductionToDemandArc
IloMSProcurementToProductionArc
IloMSProcurementToDemandArc

Date and time display

Describes how time and dates are displayed in C++ and Java™ output and in the Plant PowerOps GUI. Includes a list of all supported time zones.

In this section

Overview
An overview of the section.

C++ runtime date/time output
C++ output and time-related methods.

Java runtime date/time output
Java™ output and time-related methods.

Time zone settings
A complete listing of all available time zone settings in PPO.

Overview

Internally, Plant PowerOps handles all date/time values based on the Greenwich Mean Time (GMT) time zone. But dates are `displayed` differently, depending on the method you are using to view the output—or *plan*—that is created by the Plant PowerOps optimization engine.

- In the output of an Plant PowerOps API implementation, date/time values are always displayed as the elapsed number of `TIME_UNIT`s from the `DATE_ORIGIN` of the problem. This method of display is used in the output created by running an Plant PowerOps csv file using the `csvrun.cpp` program, or the output created by running a csv file using the `csvrun.java` program.
- When displaying dates and times from an API implementation, the time zone used depends on the method of implementation. This is discussed further in the *C++ API date/time methods* and *Java API date/time methods* sections that follow.
- In the Plant PowerOps graphical user interface (GUI), date/time values are always displayed in *local time*. That is, the date and time values are shifted plus or minus the appropriate interval from their GMT value to match the local time zone in effect on the user's computer.

C++ runtime date/time output

C++ output and time-related methods.

In this section

Overview
Presents the output from a C++ program and how to interpret the date and time data.

C++ API date/time methods
Methods of interest from IloMSDate and IloMSDay.

Overview

The output created by using the `csvrun.cpp` program to run a problem data csv file might appear like this:

```
* Start solving
*
* Time limit     : 10
*
*      no    time         tot_earliness         tot_tardiness
 total
*
*  !    1    0.0                 280.00                3990.00
4270.00
*  !    2    0.1                  80.00                2230.00
2310.00
*       3    0.1                 110.00                2220.00
2330.00
*  !    4    0.5                  80.00                2230.00
2310.00
*       5    0.5                  80.00                2230.00
2310.00
*
* Time used             : 0.551
* Nb of solutions found : 5
*
Best solution:
---------------------------------------------------------------------
DRYER                      start         end   earliness   tardiness
---------------------------------------------------------------------
TEAK03                         0          90        0.00       70.00
TEAK04                        90         180        0.00      420.00
TEAK05                       180         270       10.00        0.00
TEAK06                       270         360        0.00      560.00
PINE00                       360         430        0.00      810.00
TEAK07                       430         520       40.00        0.00
TEAK08                       520         610       30.00        0.00
PINE01                       610         680        0.00        0.00
PINE02                       680         750        0.00       90.00
TEAK09                       750         840        0.00      280.00

---------------------------------------------------------------------
tot_earliness         :        80.00
tot_tardiness         :      2230.00
total                 :      2310.00
---------------------------------------------------------------------
```

Time and date information for the solution appears in the **time** column and the **start** and **end** columns. These columns are displayed in the `TIME_UNIT` of the problem, which in this example is minutes. The value in the **start** column, for example, is the *start time* of the activity, expressed as number of `TIME_UNIT`s since `DATE_ORIGIN`, which in this example is

February 1, 2001, 00:00. Obviously, local time is not a factor when viewing Plant PowerOps output of this type.

C++ API date/time methods

There is an `IloMSDate` class available in the C++ API that can be used to create customized displays. The basic constructor information for `IloMSDate` is as follows:

```
/**
* This constructor creates a date by passing the year, month, and other time
* units as integers.
*/

IloMSDate(IloInt year, IloInt month, IloInt dayInMonth, IloInt hour=0, IloInt
 minute=0, IloInt seconds=0);

/**
* This constructor creates a date by passing a date string which complies to
* the ISO 8601 date format of &quot;YYYY-MM-DD HH::MM::SS.&quot;
*/

IloMSDate(const char* dateString);
```

You can use the C++ << operator to display the date:

```
std::ostream& operator<< (std::ostream&, const IloMSDate&);I
```

When using this method, bear in mind that the date and time returned and displayed will be in Greenwich Mean Time. Converting to local time is the developer's responsibility.

Note that when programming using the Plant PowerOps C++ API you can use another function to return the day of the week:

```
/**
* This member function returns the day in the week as an enumerated
* value: <code>IloMSSunday</code>, <code>IloMSMonday</code>, etc.
*/

IloMSDay getDayOfWeek() const;
```

One possible use of this function is when creating a break for weekends.

Java runtime date/time output

Java™ output and time-related methods.

In this section

Overview
Presents the output from a program and how to interpret the date and time data.

Java API date/time methods
Methods in the class IloMSModel to get, set and convert times and dates.

Overview

The output created by running the `api05calendar.java` program or using the `csvrun.java` program to run the `data05calendar.csv` file might appear like this:

```
* Start solving
*
* Time limit     : 10
*
*      no   time  tot_mode_cost  tot_setup_cost  tot_earliness  tot_tardiness
 tot_unperf_cost      total
*
*  !   1   0.0         1000.00          200.00         209.00       13494.00
          0.00   14903.00
*  !   2   0.1         1200.00          250.00           0.00        6921.00
          0.00    8371.00
*  !   3   0.1         1200.00          150.00          79.00        3766.00
        100.00    5295.00
*  !   4   0.2         1300.00          150.00          79.00        2926.00
        100.00    4555.00
*  !   5   0.2         1400.00          250.00         355.00        1267.00
        100.00    3372.00
*  !   6   0.2         1500.00          250.00         355.00        1057.00
        100.00    3262.00
*      7   0.2         1500.00          250.00         355.00        1057.00
        100.00    3262.00
*  !   8   0.2         1500.00          200.00         450.00         922.00
        100.00    3172.00
*      9   0.2         1500.00          200.00         450.00         922.00
        100.00    3172.00
*     10   0.3         1500.00          150.00        1089.00         742.00
        200.00    3681.00
*     11   0.3         1500.00          150.00        1089.00         742.00
        200.00    3681.00
*     12   0.3         1500.00          150.00        1114.00         707.00
        200.00    3671.00
*     13   0.3         1500.00          100.00        1109.00         707.00
        200.00    3616.00
*     14   0.3         1400.00           50.00        1673.00         707.00
        300.00    4130.00
*     15   0.3         1400.00           50.00        1673.00         707.00
        300.00    4130.00
*     16   0.3         1400.00          100.00        1034.00         752.00
        200.00    3486.00
*     17   1.3         1400.00          100.00        1034.00         752.00
        200.00    3486.00
*     18   7.0         1400.00          100.00        1034.00         752.00
        200.00    3486.00
*  !  19   7.0         1500.00          200.00         450.00         922.00
        100.00    3172.00
*     20   7.0         1400.00          100.00        1034.00         752.00
        200.00    3486.00
```

```
*  !  21   7.0        1500.00          200.00         450.00         922.00
          100.00    3172.00
*  !  22   7.0        1500.00          200.00         344.00         922.00
          100.00    3066.00
*     23   7.3        1500.00          200.00         450.00         922.00
          100.00    3172.00
*     24   7.3        1500.00          200.00         334.00        1036.00
          100.00    3170.00
*     25   7.3        1500.00          200.00         334.00        1036.00
          100.00    3170.00
*     26   7.5        1500.00          200.00         299.00         992.00
          100.00    3091.00
*     27   7.5        1500.00          200.00         299.00         992.00
          100.00    3091.00
*
* Time used             : 7.55
* Nb of solutions found : 27
*
Best solution:
-----------------------------------------------------------------------------
-------------------------
DRYER                          start        end mode  mode/setup   earliness
   tardiness unperformed
-----------------------------------------------------------------------------
-------------------------
ORDER03_TEAK_DRYING_setup          0          7    0          50         0.
00        0.00          0
ORDER03_TEAK_DRYING                7         87    1         200         0.
00       49.00          0
ORDER00_PINE_DRYING_setup         79         79    0           0         0.
00        0.00          0
ORDER00_PINE_DRYING               79        160    0         100         0.
00        0.00        100
ORDER04_TEAK_DRYING_setup         87         87    0           0         0.
00        0.00          0
ORDER04_TEAK_DRYING               87        167    1         200         0.
00      329.00          0
ORDER05_TEAK_DRYING_setup        167        167    0           0         0.
00        0.00          0
ORDER05_TEAK_DRYING              167        247    1         200        33.
00        0.00          0
ORDER06_TEAK_DRYING_setup        247        247    0           0         0.
00        0.00          0
ORDER06_TEAK_DRYING              247        327    1         200         0.
00      329.00          0
ORDER07_TEAK_DRYING_setup        354        354    0           0         0.
00        0.00          0
ORDER07_TEAK_DRYING              354        474    0         100        86.
00        0.00          0
ORDER01_PINE_DRYING_setup        474        480    0          50         0.
00        0.00          0
ORDER01_PINE_DRYING              480        550    0         100       130.
00        0.00          0
ORDER08_TEAK_DRYING_setup        550        555    0          50         0.
```

```
00       0.00           0
ORDER08_TEAK_DRYING               555         645    0         100         0.
00      35.00           0
ORDER09_TEAK_DRYING_setup         645         645    0           0         0.
00       0.00           0
ORDER09_TEAK_DRYING               645         705    1         200        95.
00       0.00           0
ORDER02_PINE_DRYING_setup         705         710    0          50         0.
00       0.00           0
ORDER02_PINE_DRYING               710         780    0         100         0.
00     180.00           0

-----------------------------------------------------------------------------
-------------------------
tot_mode_cost         :      1500.00
tot_setup_cost        :       200.00
tot_earliness         :       344.00
tot_tardiness         :       922.00
tot_unperf_cost       :       100.00
total                 :      3066.00
-----------------------------------------------------------------------------
-------------------------
```

Time and date information for the solution appears in the **time** column and the **start** and **end** columns. These columns are displayed in the TIME_UNIT of the problem, which in this example is minutes. The value in the **start** column, for example, is the *start time* of the activity, expressed as number of TIME_UNITs since DATE_ORIGIN, which in this example is February 1, 2001, 00:00. Obviously, local time is not a factor when viewing Plant PowerOps output of this type.

Java API date/time methods

The `java.util.Date` class in the Java™ API, in conjunction with the `java.text.Format`, `java.util.Calendar`, and `java.util.TimeZone` classes can be used to display in local time and with user-selectable formats for the dates.

Return type	Method
`java.util.Date`	`convertTimeToDate(int time)`
`int`	`convertDateToTime(java.util.Date date)`
`java.util.Date`	`getDateOrigin()`
`void`	`setDateOrigin(java.util.Date origin)`

The methods in the previous table are available in the class `IloMSModel`.

Time zone settings

You can select the time zone using the `TIME_ZONE` field of the `PPO_MODEL` table, or with the method `IloMSModel`::`setTimeZone`.

The list of available time zones follows, starting at GMT-12 and moving eastward.

Etc/GMT-12,

Etc/GMT-11,

MIT,

Pacific/Apia,

Pacific/Midway,

Pacific/Niue,

Pacific/Pago_Pago,

Pacific/Samoa,

US/Samoa,

America/Adak,

America/Atka,

Etc/GMT-10,

HST,

Pacific/Fakaofo,

Pacific/Honolulu,

Pacific/Johnston,

Pacific/Rarotonga,

Pacific/Tahiti,

SystemV/HST10,

US/Aleutian,

US/Hawaii,

Pacific/Marquesas,

AST,

America/Anchorage,

America/Juneau,

America/Nome,

America/Yakutat,

Etc/GMT-9,

Pacific/Gambier,

SystemV/YST9,

SystemV/YST9YDT,

US/Alaska,

America/Dawson,

America/Ensenada,

America/Los_Angeles,

America/Tijuana,

America/Vancouver,

America/Whitehorse,

Canada/Pacific,

Canada/Yukon,

Etc/GMT-8,

Mexico/BajaNorte,

PST,

PST8PDT,

Pacific/Pitcairn,

SystemV/PST8,

SystemV/PST8PDT,

US/Pacific,

US/Pacific-New,

America/Boise,

America/Cambridge_Bay,

America/Chihuahua,

America/Dawson_Creek,

America/Denver,

America/Edmonton,

America/Hermosillo,

America/Inuvik,

America/Mazatlan,

America/Phoenix,

America/Shiprock,

America/Yellowknife,

Canada/Mountain,

Etc/GMT-7,

MST,

MST7MDT,

Mexico/BajaSur,

Navajo,

PNT,

SystemV/MST7,

SystemV/MST7MDT,

US/Arizona,

US/Mountain,

America/Belize,

America/Cancun,

America/Chicago,

America/Costa_Rica,

America/El_Salvador,

America/Guatemala,

America/Managua,

America/Menominee,

America/Merida,

America/Mexico_City,

America/Monterrey,

America/North_Dakota/Cente,

America/Rainy_River,

America/Rankin_Inlet,

America/Regina,

America/Swift_Current,

America/Tegucigalpa,

America/Winnipeg,

CST,

CST6CDT,

Canada/Central,

Canada/East-Saskatchewan,

Canada/Saskatchewan,

Chile/EasterIsland,

Etc/GMT-6,

Mexico/General,

Pacific/Easter,

Pacific/Galapagos,

SystemV/CST6,

SystemV/CST6CDT,

US/Central,

America/Bogota,

America/Cayman,

America/Detroit,

America/Eirunepe,

America/Fort_Wayne,

America/Grand_Turk,

America/Guayaquil,

America/Havana,

America/Indiana/Indianapol,

America/Indiana/Knox,

America/Indiana/Marengo,

America/Indiana/Vevay,

America/Indianapolis,

America/Iqaluit,

America/Jamaica,

America/Kentucky/Louisvill,

America/Kentucky/Monticell,

America/Knox_IN,

America/Lima,

America/Louisville,

America/Montreal,

America/Nassau,

America/New_York,

America/Nipigon,

America/Panama,

America/Pangnirtung,

America/Port-au-Prince,

America/Porto_Acre,

America/Rio_Branco,

America/Thunder_Bay,

Brazil/Acre,

Canada/Eastern,

Cuba,

EST,

EST5EDT,

Etc/GMT-5,

IET,

Jamaica,

SystemV/EST5,

SystemV/EST5EDT,

US/East-Indiana,

US/Eastern,

US/Indiana-Starke,

US/Michigan,

America/Anguilla,

America/Antigua,

America/Aruba,

America/Asuncion,

America/Barbados,

America/Boa_Vista,

America/Caracas,

America/Cuiaba,

America/Curacao,

America/Dominica,

America/Glace_Bay,

America/Goose_Bay,

America/Grenada,

America/Guadeloupe,

America/Guyana,

America/Halifax,

America/La_Paz,

America/Manaus,

America/Martinique,

America/Montserrat,

America/Port_of_Spain,

America/Porto_Velho,

America/Puerto_Rico,

America/Santiago,

America/Santo_Domingo,

America/St_Kitts,

America/St_Lucia,

America/St_Thomas,

America/St_Vincent,

America/Thule,

America/Tortola,

America/Virgin,

Antarctica/Palmer,

Atlantic/Bermuda,

Atlantic/Stanley,

Brazil/West,

Canada/Atlantic,

Chile/Continental,

Etc/GMT-4,

PRT,

SystemV/AST4,

SystemV/AST4ADT,

America/St_Johns,

CNT,

Canada/Newfoundland,

AGT,

America/Araguaina,

America/Belem,

America/Buenos_Aires,

America/Catamarca,

America/Cayenne,

America/Cordoba,

America/Fortaleza,

America/Godthab,

America/Jujuy,

America/Maceio,

America/Mendoza,

America/Miquelon,

America/Montevideo,

America/Paramaribo,

America/Recife,

America/Rosario,

America/Sao_Paulo,

Antarctica/Rothera,

BET,

Brazil/East,

Etc/GMT-3,

America/Noronha,

Atlantic/South_Georgia,

Brazil/DeNoronha,

Etc/GMT-2,

America/Scoresbysund,

Atlantic/Azores,

Atlantic/Cape_Verde,

Etc/GMT-1,

Africa/Abidjan,

Africa/Accra,

Africa/Bamako,

Africa/Banjul,

Africa/Bissau,

Africa/Casablanca,

Africa/Conakry,

Africa/Dakar,

Africa/El_Aaiun,

Africa/Freetown,

Africa/Lome,

Africa/Monrovia,

Africa/Nouakchott,

Africa/Ouagadougou,

Africa/Sao_Tome,

Africa/Timbuktu,

America/Danmarkshavn,

Atlantic/Canary,

Atlantic/Faeroe,

Atlantic/Madeira,

Atlantic/Reykjavik,

Atlantic/St_Helena,

Eire,

Etc/GMT,

Etc/GMT-0,

Etc/GMT+0,

Etc/GMT0,

Etc/Greenwich,

Etc/UCT,

Etc/UTC,

Etc/Universal,

Etc/Zulu,

Europe/Belfast,

Europe/Dublin,

Europe/Lisbon,

Europe/London,

GB,

GB-Eire,

GMT,

GMT0,

Greenwich,

Iceland,

Portugal,

UCT,

UTC,

Universal,

WET,

Zulu,

Africa/Algiers,

Africa/Bangui,

Africa/Brazzaville,

Africa/Ceuta,

Africa/Douala,

Africa/Kinshasa,

Africa/Lagos,

Africa/Libreville,

Africa/Luanda,

Africa/Malabo,

Africa/Ndjamena,

Africa/Niamey,

Africa/Porto-Novo,

Africa/Tunis,

Africa/Windhoek,

Arctic/Longyearbyen,

Atlantic/Jan_Mayen,

CET,

ECT,

Etc/GMT+1,

Europe/Amsterdam,

Europe/Andorra,

Europe/Belgrade,

Europe/Berlin,

Europe/Bratislava,

Europe/Brussels,

Europe/Budapest,

Europe/Copenhagen,

Europe/Gibraltar,

Europe/Ljubljana,

Europe/Luxembourg,

Europe/Madrid,

Europe/Malta,

Europe/Monaco,

Europe/Oslo,

Europe/Paris,

Europe/Prague,

Europe/Rome,

Europe/San_Marino,

Europe/Sarajevo,

Europe/Skopje,

Europe/Stockholm,

Europe/Tirane,

Europe/Vaduz,

Europe/Vatican,

Europe/Vienna,

Europe/Warsaw,

Europe/Zagreb,

Europe/Zurich,

MET,

Poland,

ART,

Africa/Blantyre,

Africa/Bujumbura,

Africa/Cairo,

Africa/Gaborone,

Africa/Harare,

Africa/Johannesburg,

Africa/Kigali,

Africa/Lubumbashi,

Africa/Lusaka,

Africa/Maputo,

Africa/Maseru,

Africa/Mbabane,

Africa/Tripoli,

Asia/Amman,

Asia/Beirut,

Asia/Damascus,

Asia/Gaza,

Asia/Istanbul,

Asia/Jerusalem,

Asia/Nicosia,

Asia/Tel_Aviv,

CAT,

EET,

Egypt,

Etc/GMT+2,

Europe/Athens,

Europe/Bucharest,

Europe/Chisinau,

Europe/Helsinki,

Europe/Istanbul,

Europe/Kaliningrad,

Europe/Kiev,

Europe/Minsk,

Europe/Nicosia,

Europe/Riga,

Europe/Simferopol,

Europe/Sofia,

Europe/Tallinn,

Europe/Tiraspol,

Europe/Uzhgorod,

Europe/Vilnius,

Europe/Zaporozhye,

Israel,

Libya,

Turkey,

Africa/Addis_Ababa,

Africa/Asmera,

Africa/Dar_es_Salaam,

Africa/Djibouti,

Africa/Kampala,

Africa/Khartoum,

Africa/Mogadishu,

Africa/Nairobi,

Antarctica/Syowa,

Asia/Aden,

Asia/Baghdad,

Asia/Bahrain,

Asia/Kuwait,

Asia/Qatar,

Asia/Riyadh,

EAT,

Etc/GMT+3,

Europe/Moscow,

Indian/Antananarivo,

Indian/Comoro,

Indian/Mayotte,

W-SU,

Asia/Riyadh87,

Asia/Riyadh88,

Asia/Riyadh89,

Mideast/Riyadh87,

Mideast/Riyadh88,

Mideast/Riyadh89,

Asia/Tehran,

Iran,

Asia/Aqtau,

Asia/Baku,

Asia/Dubai,

Asia/Muscat,

Asia/Oral,

Asia/Tbilisi,

Asia/Yerevan,

Etc/GMT+4,

Europe/Samara,

Indian/Mahe,

Indian/Mauritius,

Indian/Reunion,

NET,

Asia/Kabul,

Asia/Aqtobe,

Asia/Ashgabat,

Asia/Ashkhabad,

Asia/Bishkek,

Asia/Dushanbe,

Asia/Karachi,

Asia/Samarkand,

Asia/Tashkent,

Asia/Yekaterinburg,

Etc/GMT+5,

Indian/Kerguelen,

Indian/Maldives,

PLT,

Asia/Calcutta,

IST,

Asia/Katmandu,

Antarctica/Mawson,

Antarctica/Vostok,

Asia/Almaty,

Asia/Colombo,

Asia/Dacca,

Asia/Dhaka,

Asia/Novosibirsk,

Asia/Omsk,

Asia/Qyzylorda,

Asia/Thimbu,

Asia/Thimphu,

BST,

Etc/GMT+6,

Indian/Chagos,

Asia/Rangoon,

Indian/Cocos,

Antarctica/Davis,

Asia/Bangkok,

Asia/Hovd,

Asia/Jakarta,

Asia/Krasnoyarsk,

Asia/Phnom_Penh,

Asia/Pontianak,

Asia/Saigon,

Asia/Vientiane,

Etc/GMT+7,

Indian/Christmas,

VST,

Antarctica/Casey,

Asia/Brunei,

Asia/Chongqing,

Asia/Chungking,

Asia/Harbin,

Asia/Hong_Kong,

Asia/Irkutsk,

Asia/Kashgar,

Asia/Kuala_Lumpur,

Asia/Kuching,

Asia/Macao,

Asia/Macau,

Asia/Makassar,

Asia/Manila,

Asia/Shanghai,

Asia/Singapore,

Asia/Taipei,

Asia/Ujung_Pandang,

Asia/Ulaanbaatar,

Asia/Ulan_Bator,

Asia/Urumqi,

Australia/Perth,

Australia/West,

CTT,

Etc/GMT+8,

Hongkong,

PRC,

Singapore,

Asia/Choibalsan,

Asia/Dili,

Asia/Jayapura,

Asia/Pyongyang,

Asia/Seoul,

Asia/Tokyo,

Asia/Yakutsk,

Etc/GMT+9,

JST,

Japan,

Pacific/Palau,

ROK,

ACT,

Australia/Adelaide,

Australia/Broken_Hill,

Australia/Darwin,

Australia/North,

Australia/South,

Australia/Yancowinna,

AET,

Antarctica/DumontDUrville,

Asia/Sakhalin,

Asia/Vladivostok,

Australia/ACT,

Australia/Brisbane,

Australia/Canberra,

Australia/Hobart,

Australia/Lindeman,

Australia/Melbourne,

Australia/NSW,

Australia/Queensland,

Australia/Sydney,

Australia/Tasmania,

Australia/Victoria,

Etc/GMT+10,

Pacific/Guam,

Pacific/Port_Moresby,

Pacific/Saipan,

Pacific/Truk,

Pacific/Yap,

Australia/LHI,

Australia/Lord_Howe,

Asia/Magadan,

Etc/GMT+11,

Pacific/Efate,

Pacific/Guadalcanal,

Pacific/Kosrae,

Pacific/Noumea,

Pacific/Ponape,

SST,

Pacific/Norfolk,

Antarctica/McMurdo,

Antarctica/South_Pole,

Asia/Anadyr,

Asia/Kamchatka,

Etc/GMT+12,

Kwajalein,

NST,

NZ,

Pacific/Auckland,

Pacific/Fiji,

Pacific/Funafuti,

Pacific/Kwajalein,

Pacific/Majuro,

Pacific/Nauru,

Pacific/Tarawa,

Pacific/Wake,

Pacific/Wallis,

NZ-CHAT,

Pacific/Chatham,

Etc/GMT+13,

Pacific/Enderbury,

Pacific/Tongatapu,

Etc/GMT+14,

Pacific/Kiritimati.

Index

A

access rights **412**
    advanced **415**
activities
    ERD **514**
activity
    definition **284**
    modeling prototypes in C++ **332**
    naming **148**
    prototype **284, 301, 356**
    valid use periods **147**
activity extent **185**
activity prototype
    definition **284**
ACTIVITY_ID **302**
activityIds method **333**
additive
    calendar intervals **138**
administrator access **412**
aggregated resources **128**
alpha service level **216**
API
    for C++, using to model and solve **324**
    for Java, using to model and solve **347**
arc
    production to demand, definition **290**
authentication **412**

B

bar
    menu **169**
batches
    cleanup after **132**
batching
    module, introduction to **70**
BATCHING_CPLEX_PARAM_ **86**
beta service level **216**
breaks
    modeling example **406**
bucket
    automatic creation of **202**
buckets
    defining **123**

C

C++
    enumerated types **349**
    modeling and solving with **324**
    modeling differences with Java **348**
    modeling example **329**
    modeling with **325**
    solving with **320**
C++ date and time display **531**
calendars
    modeling example **406**
    overlapping, additive **138**
    reasons to use **136**
    UML diagram **525**
CAPACITY **300**
capacity
    as argument **331, 355**
chm format documentation **17, 18**
CIP
    Java modeling example **381**
cleanup **132**
cleanup cost crierion **73**
cleanups
    defining need for **390**
    ERD **515**
    Java modeling example **386**
    modeling example **405**
constraints
    precedence **144**
costs
    balancing non-delivery and tardiness **276**

basic example with earliness and tardiness **275**
basic example with non-delivery **275**
courselesson1_basic.csv **312**
CPLEX parameters **86**
CPP file **339**
use for modeling and solving **324**
CPX_PARAM_ **86**
cradle manufacturing example **274**
csv
files, how to create **296, 297**
CSV file
as input, C++ **320**
as input, Java **346**
csvsch.cpp **320**
csvsch.java **346**
customize
data view **479**
Master and Transactional Data **479**
plan view **431**
cycle service level **215**
example **224**

D

dairyplant.java **367**
data model
basic **93**
extensions **119**
table of basic objects **293**
data modeling
how to **97**
data table
modify **490**
remove **489**
database
adding property columns **500**
adding tables **501**
basic usage **195**
connectivity **197**
customization **491**
loading submodel **198**
Microsoft client **495**
Oracle client **494**
saving scenario to **205**
supported **196**
table and column removal **499**
database server requirements **21, 25**
datamodelaccess.xml **480**
datamodeldisplay.xml **480**
date display **529**
C++ **531**
Java **535**
date origin **276**
days of supply
modeling example **380**
days of supply target **157**
decomposition **233**
example **235**
with Java API **241**
demand
as variable **337, 338, 360**
definition **283**
introduction **109**
uncertain **212**
variable **212**
demand coverage
material inventory **157**
DEMAND_ID **109, 304**
demands
ERD **516**
diagram
entity relationship **511**
disclaimers **19**
distribution planning **247**
example **251**
map **253**
documentation
about **15**
formats (chm, pdf, html) **17, 18**
searching **17**
due date **283**
DueTime **337, 360**
dynamic service levels **216**

E

earliness cost criterion **73**
Edit menu **170**
end max **102**
EndToEnd precedence **144**
EndToStart precedence **144**
Entity Relationship Diagram **511**
cleanups **515**
demands **516**
general overview **513**
material production flow **518**
materials and storage units **519**
procurements **520**
production orders **521**
production plans **522**
production schedule **523**
recipes, activities, modes **514**
resources **517**
setup times and costs **524**
ERD **511**
ERP **67**
error
forecasting **212**
event-based service level **216**
example
basic modeling example **271**
cradle manufacturing **274**
modeling a basic problem **298**

extending PPO GUI **420**
extension mechanism
    engine optimizer **467**
    engine optimizer class to solve data **473**
    engine optimizer factory class **469**
extension point **421, 424**
    application level **429**
    exceptions with bootstrap **425**
    to plan view **428**
extension points **432**
    plan view container configuration **441**
    plan view container defaults **448**
    plan view types **433**

F

File menu **169**
fill rate service level **215, 216**
    example **228**
firm information **118**
forecast error **212**
formats of online documentation **17, 18**

G

getActivityByIdentifier **357**
getActivityPrototype **335, 359**
getActivityPrototypeByIdentifier **333**
getCalendar **333, 357**
getDemand **337, 338, 360, 362**
getMaterial **335, 336, 359, 360**
getNumberOfProductionOrders **338, 362**
getNumberOfRecipes **335, 359**
getProductionOrder **338, 362**
getQuantity **338, 362**
getRecipe **332, 335, 337, 356, 359, 361**
getResource **333, 357**
getSetupMatrix **331, 355**
Graphical Planning Board **67**
Greenwich Mean Time **529**
GUI
    add new data table **483**

H

Help menu **172**
horizons **102**
how to model data **97**
html documentation **17, 18**

I

icon
    creating new **465**
    removing **466**
idle cost criterion **73**
IloMSMasterDataPanel **481**
IloMSModel **325, 339, 363**
IloMSOptimizerFactory **469**
ILOSTBEGIN **326**
Infinite capacity **185**
inventory **155**
    coverage corridor **157**
    days of supply target **157**
    dynamic coverage **229**
    maintaining minimal **156**
inventory cost criterion **73**
inventory deficit cost criterion **73**

J

Java
    enumerated types **349**
    modeling and solving with **347**
    modeling differences with C++ **348**
    modeling example **353**
    modeling example, dairy plant **367**
    modeling recipe constraints **401**
    modeling yogurt factory **375**
    solving with **346**
    tank modeling example **396**
    time buckets, defining **377**
    warehouse modeling **383**
Java API
    creating main program **363**
Java date and time display **535**
JDBC **493**

K

keys
    creating in mdb and csv files **297**
KPIs **73**
    customized **75**

L

layout **456**
LDAP to control access **413**
lead time
    modeling uncertain **231**
local time **529**
login entry to PPO **413**

M

magnetism **177, 185**
manifest file **424**
manufacturing plant modeling example **369**
map
    for Distribution Planning view **253**
Master Data
    customize view **479**
master planning **248**
masterdataaccess.xml **480**
masterdatadisplay.xml **480**
material **300**
    as variable **336, 360**
    discarding obsolete **150**
    introduction **103**
    shelf life and maturity **150**
material flow extent **185**

MATERIAL_ID **103**
materials
  definition **282**
  ERD **519**
  ERD of production flow **518**
  modeling intermediates **399**
  modeling intermediates Java example **378**
maturity **150**
mdb
  file, how to create **296**
menu
  creating new selection on **465**
  Edit **170**
  File **169**
  Help **172**
  removing **466**
  Tools **171**
  View **171**
  Window **172**
menu bar **169**
Microsoft Active Directory **413**
mode
  calendar **136**
  definition **287**
MODE_NUMBER **113, 302**
model
  .end **339, 363**
  example building a basic **298**
  global information **99**
model data
  how to **97**
modeling
  a basic modeling example **271**
modes
  ERD **514**
modify
  data table **490**
monitor size **21, 25**

N

naming activities **148**
navigate **177**
newActivityPrototype **332, 356**
newBreakTable **329**
newCalendar **353**
newCalendarInterval **353**
newDemand **336, 360**
newDueDate **337, 360**
newMaterial **331, 355**
newMaterialProduction **359**
newMode **333, 357**
newProdToDemandArc **362**
newProductionOrder **337, 361**
newRecipe **332, 356**
newResource **331, 355**
newResourceConstraint **333, 357**
newSetupMatrix **354**
non-delivery cost criterion **73**

O

object model
  and relational model **293**
objective
  weighted, definition **291**
objectives
  defining in an example **275**
one resource capacity **185**
online documentation **15**
optimization
  criteria **73**
  modules **70**
  profiles **76**
  scope **240**
output
  for api05calendar.cpp **341**
overview
  ERD **513**
  UML diagram **525**

P

pack activities **177**
pan tool **177**
panel
  creating a new **463**
parameters **86**
  CPLEX **86**
pdf documentation **17, 18**
persistence customization files **498**
plan
  for api05calendar.cpp **341**
plan view
  adding new **436**
  adding new configuration **443**
  configuration, removing **447**
  configurations, standard **442**
  container **445**
  example of extending **452**
  extending **432**
  extension point **448**
  removing a **440**
planner access **412**
planning
  distribution model and example **251**
  master **248**
  module, introduction to **70**
  over a distributed network **247**
  simplifying problem **89**
  simplifying with super resources **128**
planning horizon **102**
Planning Sheet
  layout **456**
plug-in **420, 421**

post processing
    introduction to **70**
PPO
    introduction to **65**
PPO_BUCKET **123**
PPO_BUCKET_TEMPLATE **202**
PPO_DEMAND
    introduction **109**
PPO_INVENTORY_MAX_COST **155**
PPO_INVENTORY_MIN_COST **156**
PPO_MATERIAL
    introduction **103**
PPO_MODEL
    introduction **99**
PPO_PRODUCTION_ORDER table **305**
PPO_RECIPE **124**
    introduction **111**
PPO_RESOURCE
    introduction **105**
PPO_RESOURCE_HIERARCHY **128**
precedence constraints **144**
printing **177**
problem decomposition **233**
processing cost criterion **73**
procurements
    ERD **520**
production network planning **247**
production order **338, 362**
    definition **289**
    ERD **521**
    how generated **70, 71**
production order extent **185**
production plans
    ERD **522**
production schedule
    ERD **523**
production to demand arc
    definition **290**
profile
    for optimization **76**
property columns **500**
publication list for PPO **15**
pull by demand decomposition **234**
push raw materials decomposition **234**

Q

QUANTITY
    in PPO_PROD_TO_DEMAND_ARC table **306**
quantity
    in SetDemands function **360**
    in setDemands function **336**
    in setProductionToDemandArcs function **338**
quantity-based service level **216**

R

recipe
    as argument **337, 361**
    defining **399**
    introduction **111**
    valid use periods **124**
RECIPE_ID **111, 301**
recipes
    definition **286**
    ERD **514**
refresh **177**
remove
    data table **489**
repair **177**
repair extent **185**
report generation **417**
REQUIRED_CAPACITY **302**
RequiredCapacity **333, 357**
RESERVOIR_ALLOWED **86**
RESERVOIR_ALLOWED parameter **86**
resource
    introduction **105**
    valid use periods **133**
resource capacity cost criterion **73**
RESOURCE_ID **105, 300**
resources
    aggregated or super **128**
    calendar **136**
    capacity, variable **136**
    cleanup **132**
    connection example **398**
    ERD **517**
    productivity, variable **136**
revenue criterion **73**

S

safety stock **219**
scenario
    saving to database **205**
schedule
    ERD **523**
schedules
    firm **118**
scheduling
    module, introduction to **70**
scheduling horizon **102**
schema, Tableau report **417**
SCM **67**
scope **240**
searching documentation **17**
service level
    cycle or basic type **216**
    dynamic **216**
    fill rate **216**
service level concepts **216**
service levels **209, 215**
SERVICE_LEVEL_TYPE **218**
SetActivities function **326, 332, 350, 356**

setBreak **353**
setBreakTable **329**
setCalendar **333, 357**
SetCalendars function **326, 329, 350, 353**
setCost **333**
SetCriterionWeights function **326, 339, 350, 363**
SetDemands function **326, 336, 350, 360**
SetDueDates function **326, 337, 350, 360**
setEarlinessVariableCost **337, 360**
setEndOfShift **353**
setFixedCost **357**
setIdentifier **332, 356**
setInitialSetupState **331, 355**
setIntDateOrigin **329, 353**
SetMaterialProductions **359**
SetMaterialProductions function **350**
SetMaterials function **326, 331, 350, 355**
setMaxBreakDuration **333, 357**
SetModel function **326, 329, 350, 353**
SetModes function **326, 332, 333, 350, 357**
setName **329, 331, 332, 333, 336, 337, 353, 355, 356, 357, 360, 361**
setPerformedStatus **332, 356**
setPeriodEndTime **353**
setPeriodicity **353**
setPeriodStartTime **353**
SetProductionOrderDemands function **326, 338, 350, 362**
SetProductionOrders function **326, 337, 350, 361**
setProductivity **353**
SetRecipeProducedMaterials function **326**
SetRecipes function **326, 332, 350, 356**
SetResources **326, 350**
SetResources function **331, 355**
setSetup **354**
SetSetupMatrices function **326, 330, 350, 354**
setSetupMatrix **331, 355**
setSetupState **332, 356**
setShiftBreakable **333, 357**
setStartMin **329, 353**
setTardinessVariableCost **337, 360**
setTimeUnit **329, 353**
setUnperformedCost **333, 357**
setup cost criterion **73**
setup times and costs
  ERD **524**
SETUP_COST **330**
SETUP_TIME **330**
setups
  modeling **130**
  secondary resources and **130**
  triangular inequality **130**
setVariableProcessingTime **333, 357**
shelf life **150**
solution
  UML diagram **525**
solve
  monitor **475**
  stopping **478**
solving
  quick path **311**
start minimum **276**
StartToEnd precedence **144**
StartToStart precedence **144**
stock corridor **209, 220**
stock coverage
  dynamic corridor **229**
stock coverage example **224, 228**
stock levels **155**
stock-out **216**
storage units
  ERD **519**
SUB_RESOURCE_ID **128**
submodel loading from database **198**
super resources **128**
SUPER_RESOURCE_ID **128**
suppliers **248**
supply network planning **247**
synchronize **177**
system requirements **21, 25**

T

table
  add a new GUI data **483**
  modify **490**
  remove **489**
Tableau **417**
tables
  adding to database **501**
tank
  modeling **396**
tanks
  modeling **403**
tardiness cost criterion **73**
target stock level **215**
template, report **417**
time bucket
  automatic creation of **202**
time buckets
  defining **123**
time definitions **276**
time display **529**
  C++ **531**
  Java **535**
time uit **276**
time zone **540**
TIME_UNIT **330**
toolbar **174**
  creating a new **463**
  creating new icon on **465**
  Gantt Diagram **176**

Plan View **176**
removing icon from **466**
Tools menu **171**
Transactional Data
customize view **479**
triangular inequality of setups **130**
type 1 service level **216**
type 2 service level **216**

U

UML diagrams **525**
calendars **525**
overview **525**
solutions **525**
units of measure
modeling example **382**
Universal Modeling Language diagrams **525**
unperformed cost criterion **73**
use period
resources **133**
user roles **412**

V

variable
demand **212**
VARIABLE_PROCESSING_TIME **302**
view
creating a new **457**
creating a new panel within existing **463**
View menu **171**
viewer access **412**
viewing access rights **415**
Visualization **67**

W

warehouses **248**
waste cost criterion **73**
waste recipes **150**
weighted objectives
definition **291**
Window menu **172**
Workload Table
layout **456**

X

XML to control access **413**

Y

yogurt manufacturing model **368**