



Cost Model Workbench User Guide

Version 2019 R2

Copyright

Copyright © 2020, aPriori Technologies, Inc. All rights reserved. aPriori Technologies, Inc., 300 Baker Avenue, Concord, MA 01742, USA.

Portions of this software were created under the direction of Professor Michael Philpott in the Department of Mechanical and Industrial Engineering at the University of Illinois at Urbana-Champaign.

Portions of this software copyright © 2000-2019 Tech Soft 3D.
Portions of this software copyright © 1995-2019 MySQL AB.
Portions of this software copyright © 2003-2019 Spatial Corporation.
Portions of this software copyright © 2006-2019 Siemens PLM.

The following trademarks and service marks are the property of aPriori Technologies, Inc.: aPriori, Cost Ticker, True Cost Convergence, Cost Insight.

Portions of this software contain copyrighted information of third parties. Title thereto is retained, and all rights therein are reserved, by the respective copyright owner. Third party software included with the product is identified below.

Document usage

This publication, as well as the software described in it, is provided under license and may only be used or copied in accordance with the terms of such license. The content of this publication is provided for informational use only. It is subject to change without notice and should not be construed as a commitment by aPriori Technologies, Inc. aPriori Technologies, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication.

Except as permitted by license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior written permission of aPriori Technologies, Inc.

The text and drawing set forth in this publication are the exclusive property of aPriori Technologies, Inc.

Third parties

Unless otherwise noted, all references to company names in sample text are designed solely to document the use of aPriori Technologies, Inc. products.

The brand names and product names used in this publication are the trademarks, registered trademarks, service marks, or trade names of their respective owners. aPriori Technologies, Inc. is not associated with any product or vendor mentioned in this publication unless otherwise noted.

This product includes additional third party software. All required third party and related software licenses and attribution are included with the aPriori software. For a complete list, see file "notices.txt" in the third-party-licenses sub-directory of the aPriori installation directory.

Software rights

The software is provided with Restricted Rights. Duplication by or disclosure to the U.S. Government is subject to the restrictions as set forth in FAR 12.212 and DFAR 227.7202 related to Commercial Computer Software and Commercial Computer Software Documentation, as applicable. Licensee agrees to always include such legends whenever software is, or is deemed to be, a deliverable under such a contract. The manufacturer is aPriori Technologies, Inc. with offices at 300 Baker Avenue, Concord, MA 01742.

Document Information

Last updated February 5, 2020

The latest version of this document can be found at the aPriori Support HelpCenter (requires registration): <https://support.apriori.com/hc>

Cost Model Workbench User Guide

About This Guide	viii
Overview	9
Related documents	9
Typographic conventions	9
Feedback and customer support	10
1 Getting Started with the Cost Model Work Bench	11
Starting and Exiting the CMWB	12
Components of the CMWB Interface	15
Working with Cost Models	16
Opening and Closing Cost Models	17
Importing and Exporting Cost Models	17
Overriding Cost Model Objects	17
Saving Cost Model Changes	17
Publishing and Reverting Cost Models	18
Working with Cost Model Data and Metadata	18
Working with Cost Model Logic	19
2 Working with Cost Model Data and Metadata	21
Navigating to Global Data	22
Navigating to the Data for a Process, Operation or Branch Node	23
Navigating from the Template Graph to the Data for a Given Node	23
Navigating from the Navigation Tree to the Data for a Given Process or Operation ..	24
Navigating from the Navigation Tree to the Data for a Given Branch Node	25
Working with Plant Variables	26
Viewing and Modifying Cost Model Variables	27
Viewing and Modifying VPE Variables	28
Creating New Plant Variables	28
Deleting Plant Variables	29
Working with Lookup Tables	29
Navigating to Lookup Tables	30
Viewing and Modifying Lookup Tables	32
Viewing and Modifying Lookup Table Definitions	33
Creating Lookup Table Definitions	34
Deleting Lookup Table Definitions	35
Adding New Lookup Tables	35
Removing Lookup Tables	36
Working with Machine Metadata	36
Viewing and Modifying Machine Types	37
Adding Machine Types	39
Deleting Machine Types	39
Adding or Modifying a Machine Field for Multiple Processes	40

Working with Material Metadata	43
Viewing and Modifying Material Types	44
Adding or Modifying a Material Field for Multiple Process Groups	45
Working with Material Stock Metadata	48
Viewing and Modifying Material Stock Types	49
Working with Tool Shop Metadata	50
Viewing and Modifying Tool Shop Types	51
Adding Tool Shop Types	52
Deleting Tool Shop Types	53
Associating a Tool Shop Type with a Process	53
Removing a Tool Shop Type from a Process	53
Working with Tool Material Metadata	54
Viewing and Modifying Tool Material Types	55
Adding Tool Material Types	56
Deleting Tool Material Types	56
Working with Process Setup Options	57
Navigating to Process Setup Options	58
About Process Setup Options	59
Modifying Process Setup Options	60
Adding and Deleting Process Setup Options	61
Including and Excluding GCD Types	62
Working with Column Groups and Column Properties	63
Working with Node Attributes	65
Navigating to Node Attributes	65
Adding, Removing, and Modifying Node Attributes	66
Predefined Node Attributes	66
3 Working with Cost Model Logic	79
CSL Language Overview	80
Formulas	81
Rules	82
Advice Rules	82
Imports	84
Values	85
Expressions	85
Line Continuation	90
Comments	90
Searching Within and Across CSL Modules	91
Using the CSL Debugger	94
CSL Reference Information	108
Viewing and Editing CSL Modules	109
Navigating to Global CSL Modules	109
Navigating to the CSL Modules for a Given Node	110
Viewing CSL Modules	112
Editing CSL Modules	113

Adding CSL Modules.....	116
Deleting CSL Modules.....	117
Creating and Deleting Processes, Operations, and Branch Nodes	117
Creating and Copying Processes	118
Creating and Copying Operations	123
Creating Branch Nodes.....	126
Deleting Processes, Operations, and Branch Nodes	127
Working with Formula Tables	128
Adding a Formula to the Formula Table	128
Controlling Whether a Custom Output Appears in the Part Details Tab	130
Template Pruning	131
Context of Evaluation.....	131
Example	132
Material Stock Selection	132
Context of Evaluation.....	133
Finding or Designating a Routing's Stock Selector	133
Stock Selector Input	134
Stock Selector Outputs.....	134
Example	135
Process and Operation Optionality	136
Context of Evaluation.....	136
Example	137
Process and Operation Feasibility	140
Context of Evaluation.....	141
Examples	141
Machine Selection	143
Context of Evaluation.....	144
Example	144
Tool Selection.....	148
Process and Operation Taxonomy.....	148
Context of Evaluation.....	149
Example	149
Working with Zero-or-More Nodes	151
Working with Templates.....	153
About Templates.....	153
Viewing and Editing Templates.....	155
4 Cost Engine Details	157
Hierarchies	158
Algorithm Overview.....	159
Process Template Expansion and Pruning	160
Material Stock Selection	160
Feasibility and Machine Selection.....	162
Operation Assignment	163

Operation Costing	166
Process Costing	167

5 Cost Scripting Language Reference 168

Overview	169
Module Types.....	169
Module Contents.....	170
Values and Expressions	171
Modules, Inputs, and Outputs	172
Syntax.....	173
Inputs	173
Outputs.....	183
Return Values.....	185
Imports	186
Formulas and Rules.....	187
Formulas.....	187
Set Blocks	188
Rules.....	189
Messages.....	190
Advice Rules	190
Advice.....	190
Function Definitions.....	191
Expressions	192
Arithmetic Expressions.....	192
String Expressions	193
Boolean Expressions.....	193
Conditional Expressions	194
Function Invocations.....	194
Query Expressions.....	195
Foreach Expressions.....	197
Like Expressions	199
Identifiers and Literals.....	200
Simple Identifiers	200
Complex Identifiers	200
Numerical Literals	200
String Literals.....	201
Boolean Literals.....	201
Comments and Line Continuation.....	201
Predefined Functions.....	202
Numeric Functions	202
String Functions.....	204
List Functions.....	206
Map Functions.....	207
Node Attribute Functions.....	208
Routing Navigation Functions	212

Error Handling Functions.....	212
Interpolation Function	213
Miscellaneous Functions.....	214
6 Common Task Examples	227
Adding a New Process to a Process Group	228
Creating a New Process from Scratch	228
Creating a New Process from an Existing Process	228
Copying the Process.....	229
Renaming the Operations	230
Adding Operations	231
Adding New Processes and Operations to Templates	232
Navigating to and Modifying Templates.....	232
Modifying the Process-level Routings.....	233
Modifying the Operation-level Routings.....	234
Defining and Modifying Machine Types	235
Defining a Machine Type for a New Process.....	235
Modifying a Machine Type—Padding Cycle Time.....	239
Modifying a Machine Type—Preferring One Class of Machines Over Another.....	240
Modifying Machine Selection	241
Removing unwanted machine checks.....	241
Preferring One Class of Machines Over Another	244
Adding Feasibility Rules.....	245
Adding New Plant Variables	247
Adding Plant Variables for a New Process	247
Adding Plant Variables—Padding Cycle Time	249
Adding New Process Setup Options	250
Adding a Setup Option—Using CSL, Formula, and User Modes	250
Adding a Setup Option—Using List Mode to Access a Lookup Table	252
Adding a Setup Option—Padding Cycle Time	253
Adding Lookup Tables	255
Adding a Lookup Table Definition	255
Adding a Lookup Table	256
Modifying Taxonomy Modules	258
Modifying the Formula Table.....	258
Modifying the Cycle Time Formulas.....	261
Padding Cycle Time by Adding a Constant	263
Adding and Modifying Library Modules	265
Creating a New Library.....	265
Modifying a Library Module.....	266

About This Guide

This section provides information about this User Guide, and the other ways in which aPriori supports the aPriori application.

Key topics include:

- Overview
 - Related documents
 - Typographic conventions
 - Feedback and customer support
-

Overview

This User Guide contains detailed information about using aPriori's Cost Model Work Bench (CMWB) to customize cost models.

Related documents

In addition to this guide, you can find more information about the aPriori application in the following documents:

- *aPriori User Guide* – This guide contains detailed information about the aPriori solution. It is designed as a reference for your everyday work.
- *aPriori Cost Model Guide* – This guide contains detailed information about process groups and includes a chapter on direct and indirect overhead. **Note:** This is a new document as of 2015 R1 SP1 and contains chapters that formerly appeared in the *aPriori User Guide*.
- *aPriori System Administration Guide* – This guide contains detailed information about administering the aPriori solution using the System Admin toolset. It is designed as a reference for aPriori system administrators.
- *aPriori VPE Administration Guide* – This guide contains detailed information about using the tools in the virtual production environment (VPE) toolset to maintain the VPEs in your aPriori deployment. It is designed as a reference for VPE administrators.
- *Release Notes* – This document highlights the changes made in aPriori since the previous release. It also contains last minute information about the release.
- *Installation Guide* – This guide contains detailed information about installing aPriori.
- *System Requirements* – This document provides information on the minimum and recommended client and server requirements to run aPriori, as well as the CAD file formats supported by aPriori.

Typographic conventions

The following conventions are used in this guide to convey additional information.

Style	Description	Example
Code	Code style is used for text that is used literally, appearing exactly as shown. This includes command names, path and file names, and system information.	E:\setup.exe
<i>Italic code</i>	Italic code style is used for names of variables that you must provide. For example, you need	C:\aPriori\your_file

	to supply a value for <i>your_file</i> in the path name example to the right.	
GUI	GUI style is used to indicate objects in the aPriori interface.	the Document field
GUI Action	GUI Action style is used to indicate objects in the aPriori interface that you click, select, or otherwise act upon.	Click OK .

Note Notes highlight information, provide supplementary information, offer time-saving or easier ways to perform the same task, or explain how to prevent errors or data loss. Be sure to read this information carefully.

Feedback and customer support

We appreciate your comments about this guide. Please contact us with your comments, questions, and requests for technical support.

Website: <http://www.apriori.com/support>

Email: support@apriori.com

1 Getting Started with the Cost Model Work Bench

This guide tells you how to use aPriori's Cost Model Work Bench (CMWB) to customize cost models. It covers how to navigate the resources made available by the CMWB in order to view and modify cost model data, metadata, and logic. It covers the syntax and semantics of Cost Scripting Language (CSL), as well as CSL predefined functions and constants. It also describes how the cost engine, templates, and GCD hierarchy determine the flow of CSL evaluation. Finally, it provides examples of common customization tasks.

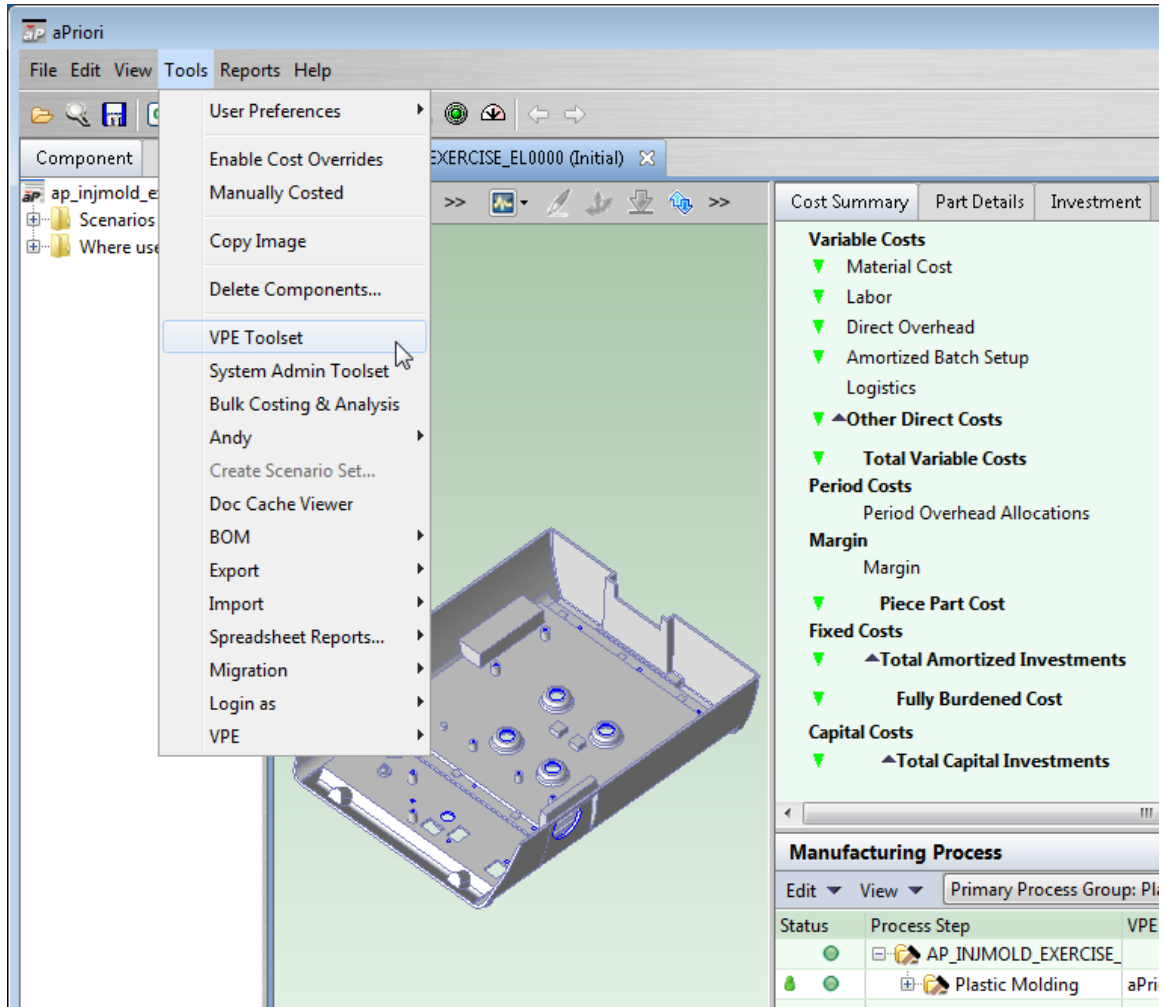
This chapter includes the following topics:

- Starting and Exiting the CMWB
 - Components of the CMWB Interface
 - Working with Cost Models
 - Working with Cost Model Data and Metadata
 - Working with Cost Model Logic
-

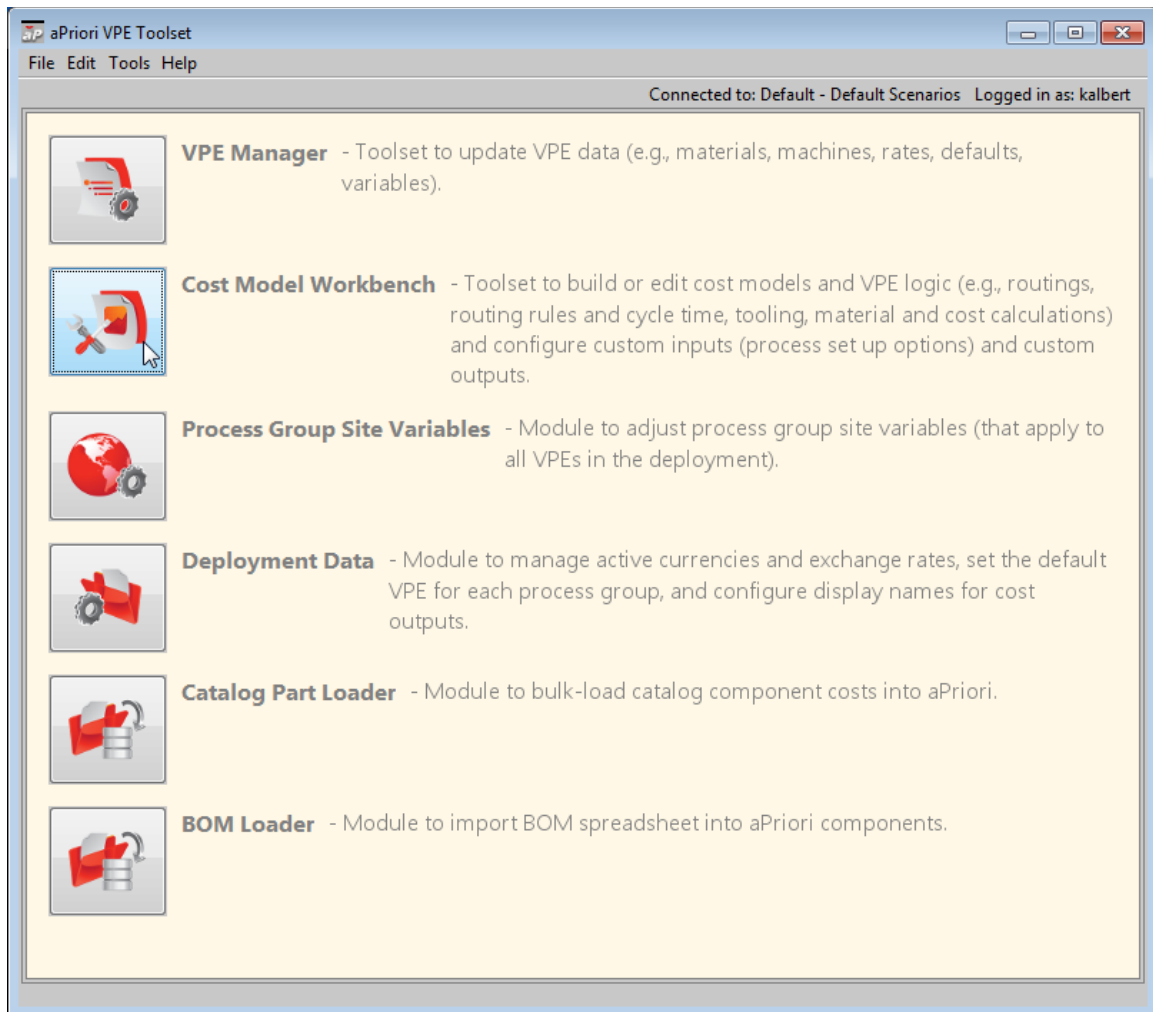
Starting and Exiting the CMWB

Follow these steps to start the CMWB:

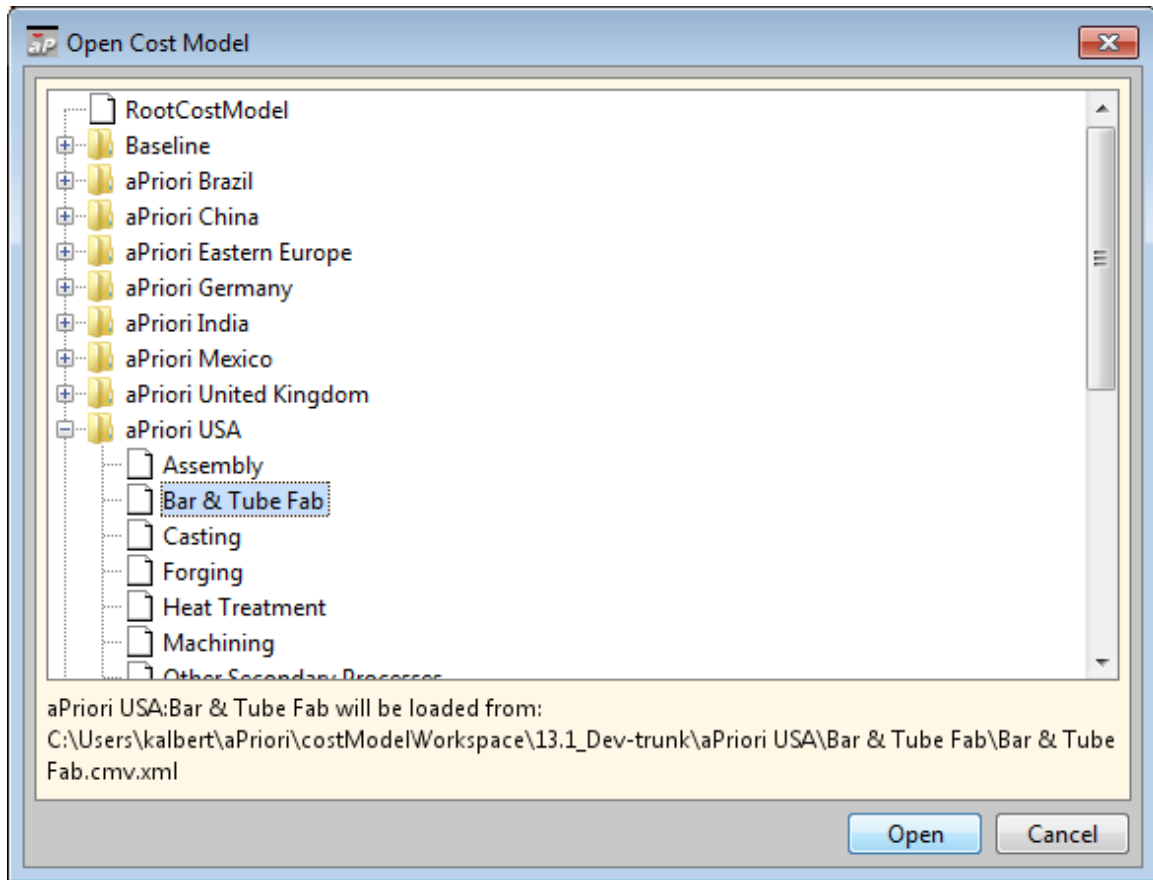
- 1 In aPriori Desktop, select **VPE Toolset** from the **Tools** menu.



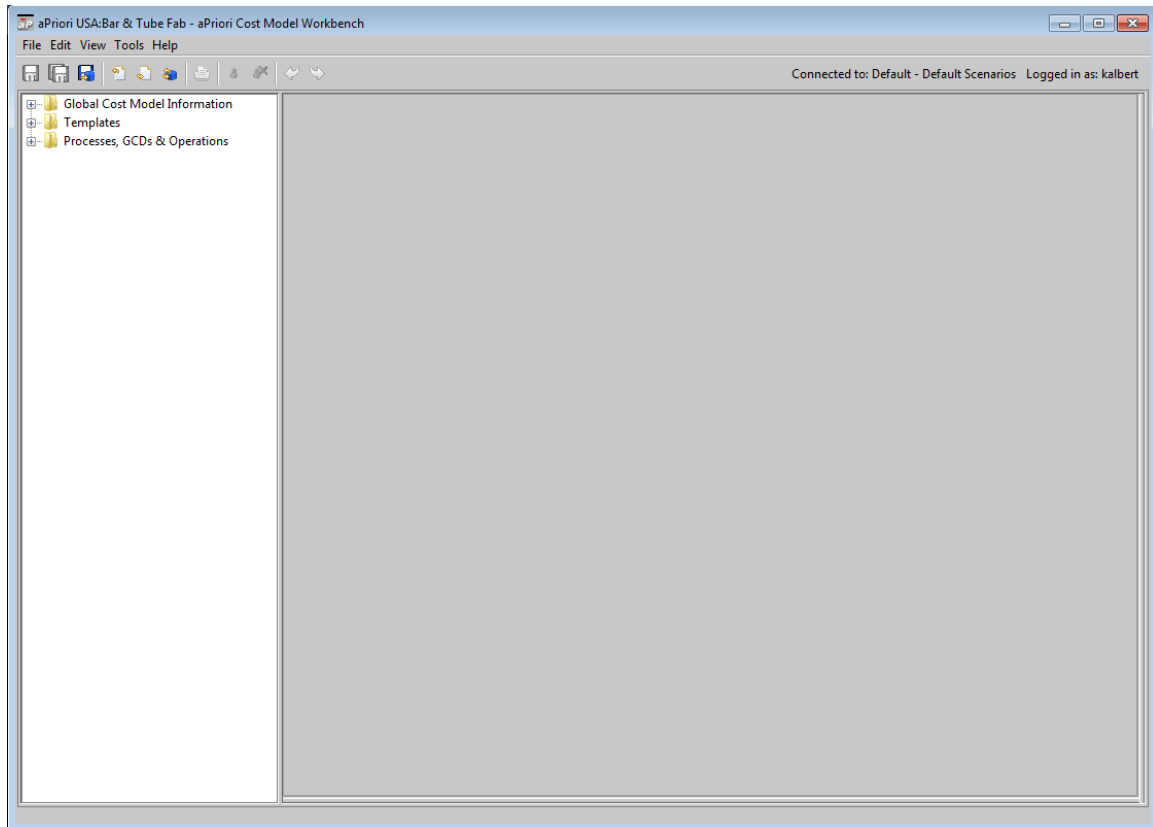
- 2 In the aPriori VPE Toolset window, click the icon next to **Cost Model Workbench**.



- 3 In the Open Cost Model dialog, expand the VPE that contains the cost model you want to open, select the desired cost model, and click **OK**.



The CMWB window appears.



To exit the CMWB, select **Exit** from the **File** menu.

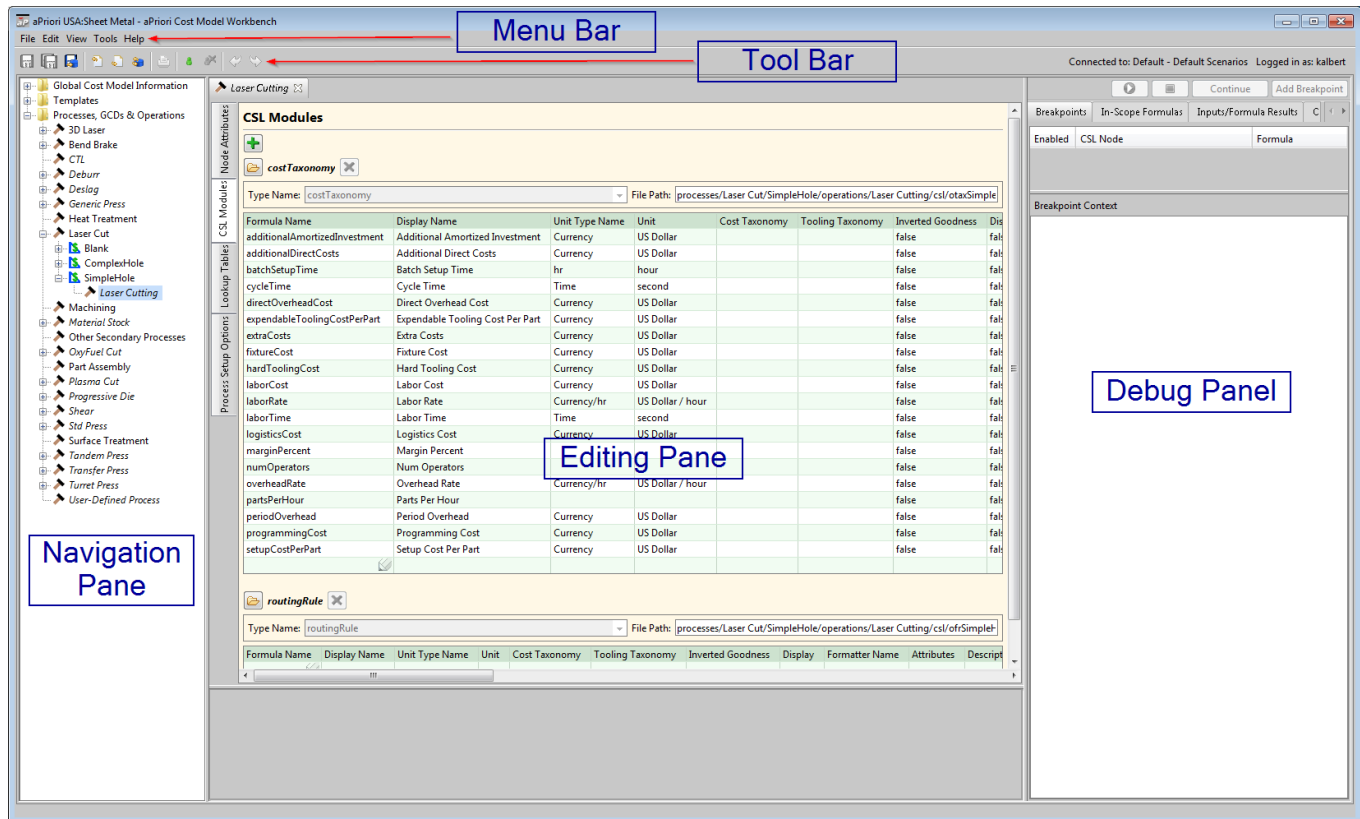
Components of the CMWB Interface

The instructions in this guide refer to the following major elements of the CMWB graphical user interface:

- **Menu bar:** provides the standard set of menus: **File**, **Edit**, **View**, **Tools**, and **Help**.
Note: Menu items which are useful in both the Cost Model Workbench and the VPE Administration Toolset (such as **Tools > Search CSL**) are documented in the *VPE Administration Guide*. See that document for more information.
- **Tool bar:** displays tool icons that provide shortcuts for operations such as **Save**, **Override Object**, and **Publish Cost Model**.
- **Navigation pane:** provides an expandable/collapsible navigation tree with three top-level nodes:
 - **Global Cost Model Information:** provides access to global data, metadata, and logic for the current cost model.
 - **Templates:** provides access to the template for each GCD type associated with the current cost model, as well as to the local data, metadata, and logic associated with branch nodes of those templates.
 - **Processes, GCDs & Operations:** displays the process-GCD-operation hierarchy, and provides access to the data, metadata, and logic for each process and operation associated with the current cost model.

- Editing pane: editable display that provides the content of global or node-specific data, metadata, or logic, including formula tables, metadata tables (such as machine type tables), and CSL module content.
- Debug panel: provides the interface to the CSL debugger, which allows you to set breakpoints, access in-scope formula and input values, and examine evaluation contexts. Use the **View** menu in the menu bar to show and hide the debug panel.

The following figure shows these elements:



Working with Cost Models

Most of the tasks described in the guide involve customizing a cost model. You customize a cost model as part of the customization of a VPE. When you customize a VPE, you generally start by creating, in a staging environment, a copy of another VPE from which you want the new VPE to inherit. See the VPE Administration Guide for information on exporting VPEs from a production environment and importing them into a staging environment, as well as for information on copying VPEs.

Cost model customizations fall into two categories:

- Data or metadata customization: includes plant variable and lookup table customization, machine, material, and tool shop metadata customization, as well as customization of process setup options. See [Working with Cost Model Data and Metadata](#).

- Logic customization: includes customization of templates that control process and operation routing, as well as customization of CSL code that controls the evaluation of taxonomy formulas, routing rules, and machine selection logic. See [Working with Cost Model Logic](#).

In addition to customization of a cost model's individual elements (such as metadata tables and CSL modules), you can also use the CMWB to perform a few operations on cost models as a whole:

- Opening and Closing Cost Models
- Importing and Exporting Cost Models
- Overriding Cost Model Objects
- Saving Cost Model Changes
- Publishing and Reverting Cost Models

Opening and Closing Cost Models

When you start the CMWB, you specify a cost model to open. If you want to open a different cost model, follow these steps:


- 1 Select **Open Cost Model...** from the **File** menu. The Open Cost Model dialog appears.
- 2 Expand the VPE that contains the cost model you want to open, select the desired cost model, and click **OK**.

To close a cost model, select **Close Cost Model** from the **File** menu.


Importing and Exporting Cost Models


The CMWB provides import/export functionality in order to support integration of cost models with source control systems. Contact aPiori Professional Services for information on performing this type of integration.

Overriding Cost Model Objects

To modify a cost model's data, metadata, or logic, you sometimes must override a cost model object such as data table, CSL module, or process setup option. If an object is inherited, before you modify it, select **Override Object** from the **Edit** menu in the menu bar, or click the override icon, , in the toolbar.

Saving Cost Model Changes


To save changes to an individual cost model object (such as a table, module, or process setup option), select **Save** from the **File** menu in the menu bar, or click  in the toolbar.

To save all outstanding changes to all unsaved objects, select **SaveAll** from the **File** menu in the menu bar, or click  in the toolbar.

Publishing and Reverting Cost Models

When you make changes to a cost model, such as changes to data tables, PSOs, or CSL modules, you can save the changes with the **Save** item on the **File** menu. This saves the changes in the CMWB's local copy. A separate step publishes the changes to the associated aPriori database.

To publish changes to the database, select **Publish Cost Model** from the **File** menu, or click  in the toolbar.

After you have saved changes locally, but before you have published them, you can roll back those changes and revert your cost model to its last published state by selecting **Revert to Database** from the **File** menu, or clicking  in the toolbar

Working with Cost Model Data and Metadata

aPriori costs a part based on the part's geometry, end-user-specified production information, and the data and logic specified by a cost model. The CMWB allows you to manage cost model metadata, such as the tables that define machine types, material types, and lookup table definitions. This metadata includes attribute names, value types, default values, and so on.

The CMWB also allows you to customize some types of data, including plant variables and lookup tables. In many cases, you must use the VPE Manager tool in order to customize the data whose corresponding metadata is managed by the CMWB. For example, machine and material metadata is managed by the CMWB, but you must use the VPE manager to manage the data for individual machines and materials.

The next chapter, *Working with Cost Model Data and Metadata* covers data and metadata customization. It includes the following topics:

- **Navigation:** describes how to navigate to global data and metadata, as well as how to navigate to process-specific and operation-specific data and metadata. This topic also covers how to navigate to data and metadata associated with branch nodes. See [Navigation](#).
- **Navigating to Global Data and Navigating to the Data for a Process, Operation or Branch Node.**
- **Plant variables:** covers how to view and modify plant variables (also known as cost model variables), as well as how to create and delete plant variables. This section also describes how to access plant variables in CSL. See [Working with Plant Variables](#).
- **Lookup tables:** describes how to navigate to global and local lookup tables, and how to modify, add, and remove lookup tables. In addition, this section covers how to modify, create, and delete lookup table definitions (that is, lookup table metadata). It also describes how to access lookup table data in CSL. See [Working with Lookup Tables](#).

- Machine metadata: covers how to view and modify machine types, as well as how to create and delete machine types. This section also describes how to access machine data in CSL. See [Working with Machine Metadata](#).
- Material and stock metadata: describes how to view and modify material types and material stock types, as well as how to access material and stock data in CSL. See [Working with Material Metadata](#) and [Working with Material Stock Metadata](#).
- Tool shop and tool material metadata: covers how to view, modify, add, and delete tool shop types and tool material types, as well as how to associate a tool shop type with a process and disassociate it from a process. This topic also describes how to access tool shop and tool material data in CSL. See [Working with Tool Shop Metadata](#) and [Working with Tool Material Metadata](#).
- Process setup options: describes how to navigate to and modify the process setup options for a given process, operation, or branch node. In addition, this section covers how to add and delete process setup options. It also describes how to access the current value and mode of a process setup option in CSL. See [Working with Process Setup Options](#).
- GCD types: covers how to control which GCD types aPriori attempts to recognize for a given process group. See [Including and Excluding GCD Types](#).
- Column properties and column groups: describes how to control the appearance of data in VPE Manager and in aPriori end user tables such as material and machine tables. See [Working with Column Groups and Column Properties](#).

Examples of working with cost model data can be found in the chapter [Common Task Examples](#).

Working with Cost Model Logic

You customize cost model logic by working with templates and CSL modules. This type of customization is covered in the chapter [Working with Cost Model Logic](#). The chapter covers the following topics:

- CSL language and debugger: provides an introduction and overview for the CSL language, including module types, formulas, rules, values, and expressions. This section also covers the CSL debugger. See [CSL Language Overview](#). This guide also includes a [Cost Scripting Language Reference](#).
- CSL modules: covers viewing and editing CSL modules, as well as creating and deleting CSL modules. See [Viewing and Editing CSL Modules](#).
- Nodes: describes creating, copying and deleting processes, operations, and branch nodes. See [Creating and Deleting Processes, Operations, and Branch Nodes](#).
- Module types: describes each module type's purpose, behavior, and context of evaluation. These sections also provide sample code from each module type, together with an explanation of the various CSL constructs employed. The module types are covered in the following sections:
 - [Template Pruning](#)
 - [Material Stock Selection](#)

- Process and Operation Optionality
- Process and Operation Feasibility
- Machine Selection
- Tool Selection
- Process and Operation Taxonomy
- Templates: covers the purpose of templates and the different kinds of template nodes. This section also describes the graphical and textual syntax and semantics of templates, as well as how to view and edit templates. See [Working with Templates](#).

Each section in this chapter includes one or more examples. Additional examples of working with cost model logic can be found in the chapter Common Task Examples.

This guide also includes a chapter, Cost Engine Details, which describes the high-level logic shared by all cost models. It details how templates together with a part's GCD hierarchy determine the flow of evaluation of the various types of CSL modules.

2 Working with Cost Model Data and Metadata

aPriori costs a part based on the logic provided by the cost engine, templates, and CSL modules, together with the data provided by the part's geometry, production information (such as material selection), and cost model data (such as machine and material attributes). This chapter covers the management of cost model data with the CMWB, including management of cost model metadata, which provides the schema or definition (attribute names, types, default values, and so on) for cost model data.

This chapter includes the following topics:

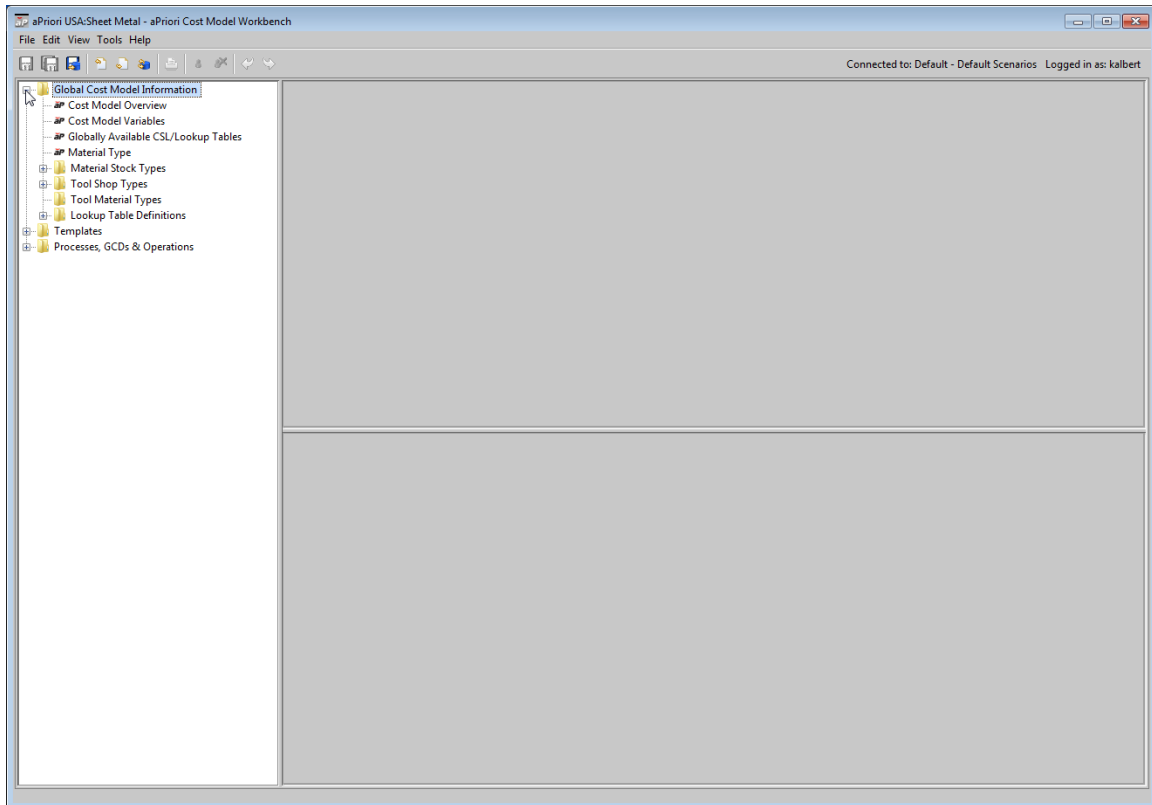
- Navigating to Global Data
 - Navigating to the Data for a Process, Operation or Branch Node
 - Working with Plant Variables
 - Working with Lookup Tables
 - Working with Machine Metadata
 - Working with Material Metadata
 - Working with Material Stock Metadata
 - Working with Tool Shop Metadata
 - Working with Tool Material Metadata
 - Working with Process Setup Options
 - Including and Excluding GCD Types
 - Working with Column Groups and Column Properties
-

Navigating to Global Data

A cost model's data is either global to the cost model or associated with a specific process, operation, or branch node.

Follow these steps to display information about and links to the global data for the current cost model:

- 1 In the CMWB navigation pane, expand the **Global Cost Model Information** node.



- 2 Double click or expand one of the following:
 - **Cost Model Variables:** See [Working with Plant Variables](#).
 - **Globally Available CSL/Lookup Tables.** See [Working with Lookup Tables](#).
 - **Material Type.** See [Working with Material Metadata](#).
 - **Material Stock Types.** See [Working with Material Stock Metadata](#).
 - **Tool Shop Types.** See [Working with Tool Shop Metadata](#).
 - **Tool Material Types.** See [Working with Tool Material Metadata](#).
 - **Lookup Table Definitions.** See [Viewing and Modifying Lookup Table Definitions](#).

Navigating to the Data for a Process, Operation or Branch Node

Data that is not global (see Navigating to Global Data) is associated with a single template node. (Note that same template node, in this sense of node, can appear in multiple templates.) There are two general ways to access the data associated with a given template node: from the template graph below the editing pane, and directly from the navigation pane.

This section describes the following tasks:

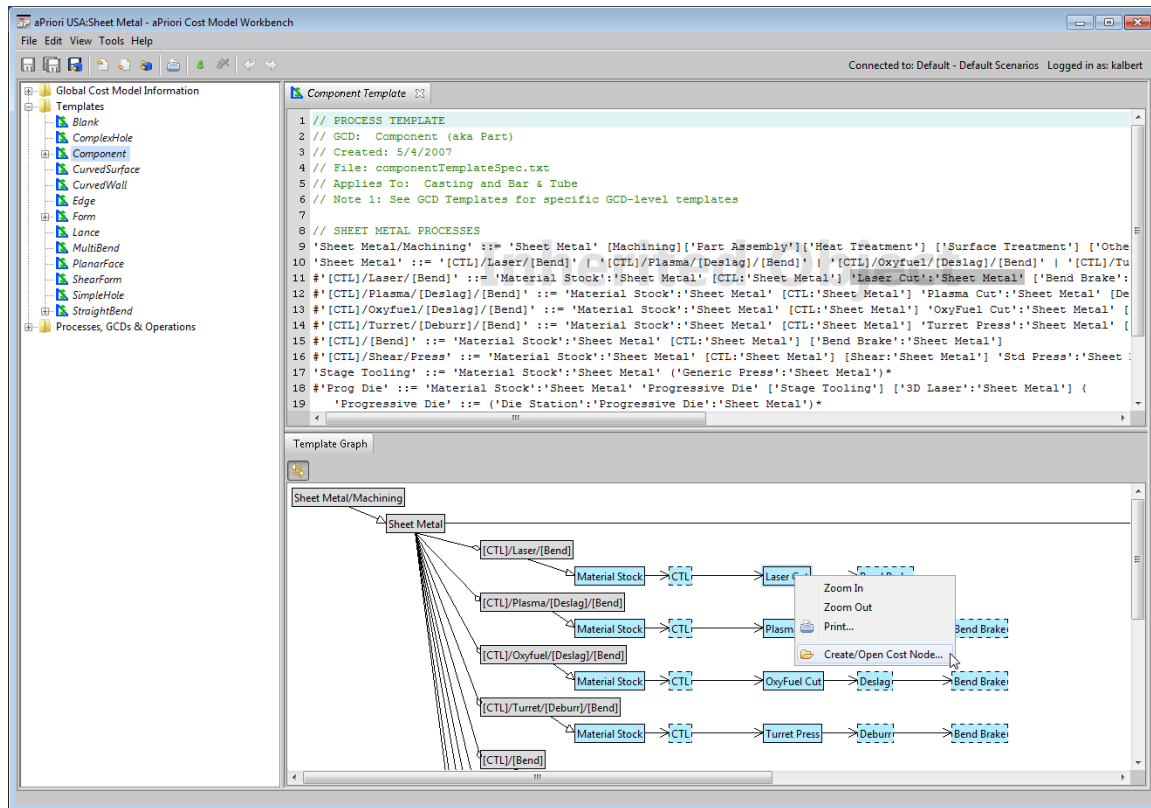
- Navigating from the Template Graph to the Data for a Given Node
- Navigating from the Navigation Tree to the Data for a Given Process or Operation
- Navigating from the Navigation Tree to the Data for a Given Branch Node

Navigating from the Template Graph to the Data for a Given Node

Caution: Use this navigation method only to navigate to processes, operations, or branch nodes that you are certain exist in the navigation pane. Some template nodes exist only in a template (for organizational purposes) and have no associated data or logic in the cost model. Using this method on such a node unnecessarily creates an empty node in the navigation pane and clutters the cost model.

To access, from a template graph, the modules for a given node, follow these steps:

- 1 In the navigation pane, expand **Templates**.
- 2 Double click a GCD type in whose template the desired node occurs. The template specification appears in the editing pane, and the template graph appears in the pane below it.



- 3 In the template graph, right click on the desired node, and select **Create/Open Cost Node**.
- 4 In the editing pane, select one of the following tabs:
 - **Lookup Tables**. See [Working with Lookup Tables](#).
 - **Process Setup Options**. See [Working with Process Setup Options](#).
 - **Machine Type**. See [Working with Machine Metadata](#).
 - **Tool Shop Type**. See [Associating a Tool Shop Type with a Process](#).

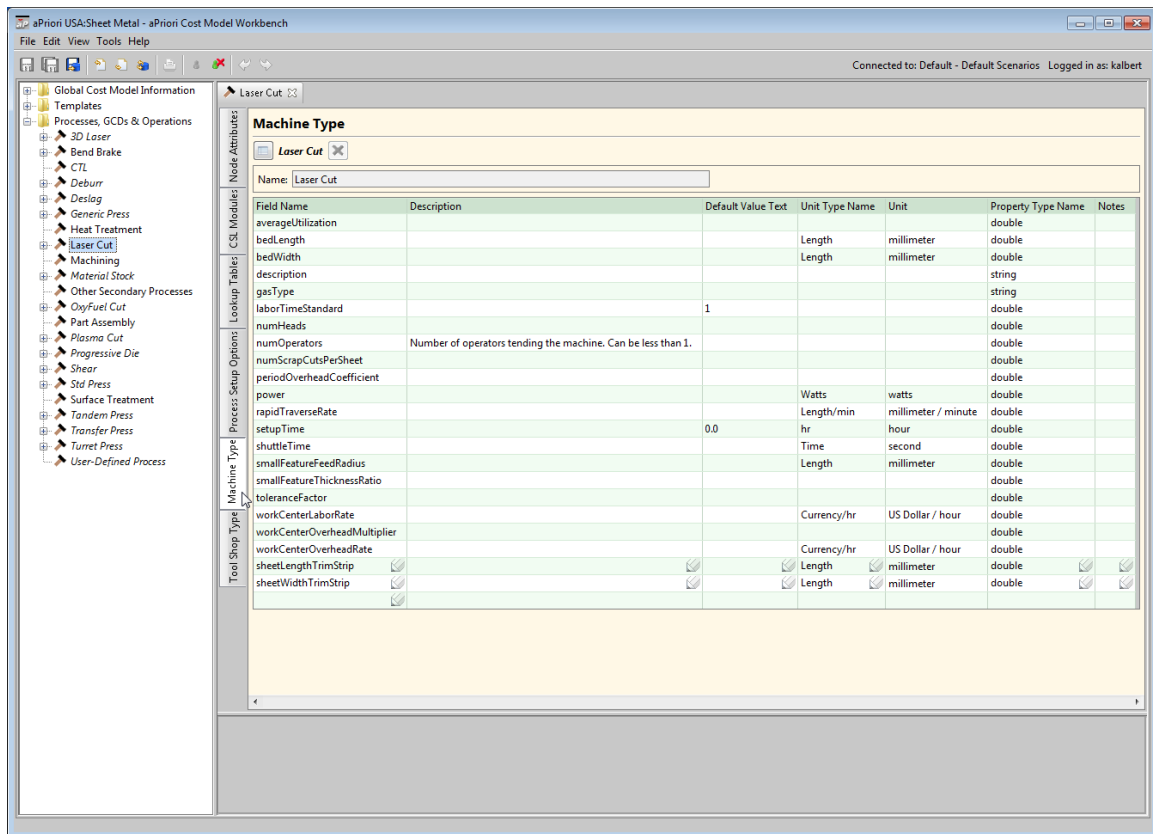
Navigating from the Navigation Tree to the Data for a Given Process or Operation

For a node that represents a process or operation (as opposed to a branch node), you can access the node's modules directly from the navigation pane as follows:

- 1 In the navigation pane, expand **Processes, GCDs & Operations**.
- 2 If the node represents a process, double click the process that the node represents. Otherwise, expand a process that can serve as an ancestor (in the process-operation hierarchy—see [Cost Engine Details](#)) of the desired operation.
- 3 Expand a GCD type to whose creation the desired operation can contribute.
- 4 Double click the desired operation under the expanded GCD, if it appears. Otherwise, expand an operation that can serve as ancestor (in the process/operation

hierarchy) of the desired operation, and go back to step 3 to continue down the hierarchy.

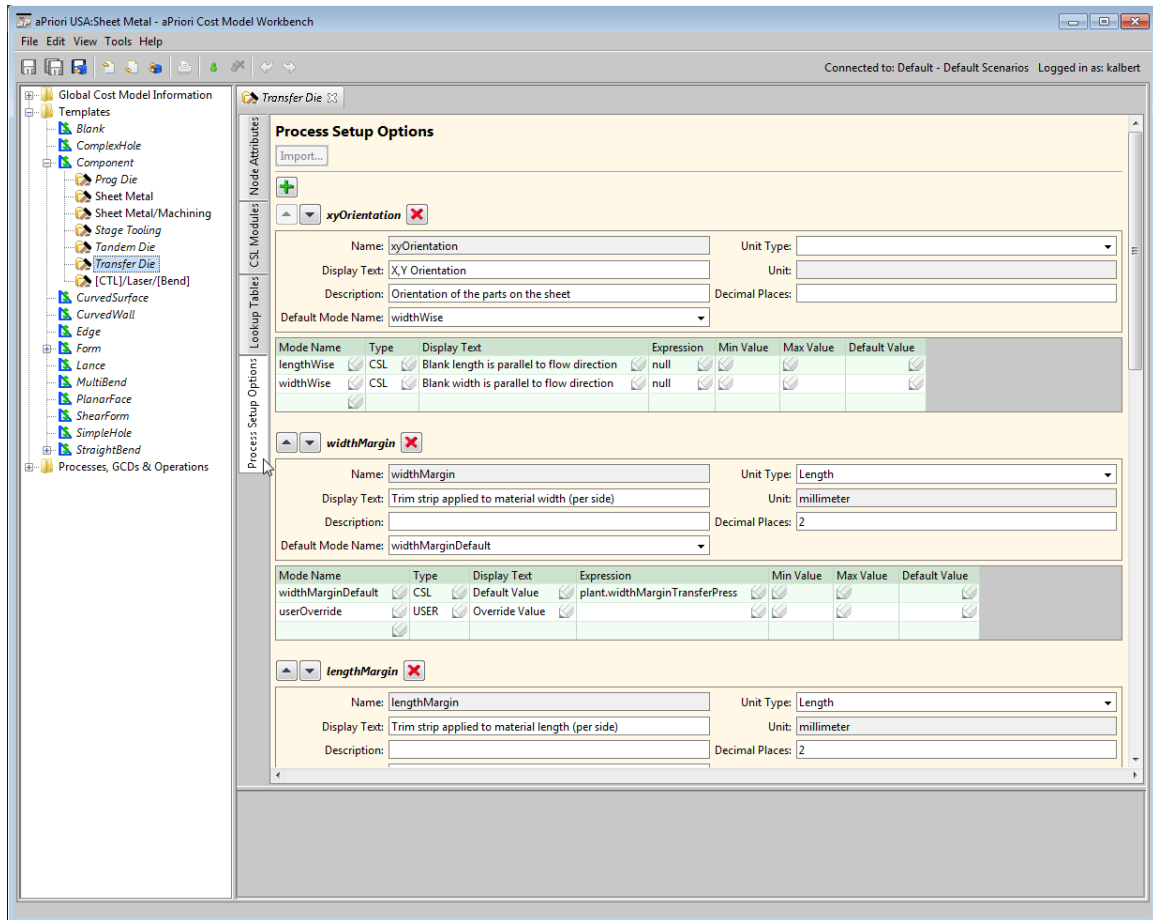
- 5 In the editing pane, select one of the following tabs:
 - **Lookup Tables.** See [Working with Lookup Tables](#).
 - **Process Setup Options.** See [Working with Process Setup Options](#).
 - **Machine Type.** See [Working with Machine Metadata](#).
 - **Tool Shop Type.** See [Associating a Tool Shop Type with a Process](#).



Navigating from the Navigation Tree to the Data for a Given Branch Node

For branch nodes, you can access a module for a given node directly from the navigation pane as follows:

- 1 In the navigation pane, expand **Templates**.
- 2 Expand a GCD type in whose template the desired node occurs.
- 3 Double click the desired node under the expanded GCD.
- 4 In the editing pane, select one of the following tabs:
 - **Lookup Tables.** See [Working with Lookup Tables](#).
 - **Process Setup Options.** See [Working with Process Setup Options](#).



Working with Plant Variables

There are two kinds of plant variables:

- Cost model variables: provide data that is global to a given cost model in a given VPE
- VPE variables: provide data that is global to a given VPE

CSL code can access plant variables through fields of the CSL standard input `plant`. Here is an example:

```
requiredShotSize = plant.shotSizeSafetyFactor * _
    (part.volume * numCavities *
    plant.densityPolystyreneForShotSize)/1000000
```

See [CSL Language Overview](#) for more information on CSL.

The CSL field names, such as `shotSizeSafetyFactor` and `densityPolystyreneForShotSize`, are specified in the **Variable Name** field of the Cost Model Variables table (see [Viewing and Modifying Cost Model Variables](#)).

This section describes the following tasks:

- Viewing and Modifying Cost Model Variables
- Creating New Plant Variables
- Deleting Plant Variables

Viewing and Modifying Cost Model Variables

To view and modify cost model variables, follow these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information**, and double click **Cost Model Variables**. The Cost Model Variables table appears in the editing pane.


Variable Name	String Value	Unit Type Name	Unit	Notes
Inherited:				
applyLabelRate	5	Time	second	Rate at which an operator can apply one label.
applyMoldReleaseRate	371612			Rate at which a mold release agent is applied to the mold surface.
betweenPartsMoldBorder	76.2	Length	millimeter	3" between parts in the mold
betweenPartsMoldBorder150T	38.1	Length	millimeter	1.5" between parts in the mold if press tonnage < 150T
betweenPartsMoldBorder300T	50.8	Length	millimeter	2" between parts in the mold if press tonnage < 300T
betweenPartsMoldBorder500T	76.2	Length	millimeter	3" between parts in the mold if press tonnage > 300T
cavityPlateDepthBorder	152.4			make 6" also, same as core border
clampForceSafetyFactor	1.1			multiplier of clamp pressure required to insure that the machine is capable of applying the
clampSpacing	127			Default distance between adjacent clamps along the mold's parting line.
clampingRate	2			Rate at which a clamp can open or close. Determines the open and close mold cycle time.
coarseTextureCostperSqM	15500			Per square meter, from \$10 per mold area square inch to texture the surface
colorantCost	0.75	CostPerMass	US Dollar / kilogram	Cost of colorant per unit mass. The colorant cost is multiplied by the rough mass of the p
coreInstallRate	15	Time	second	Rate at which an operator can install one core.
corePlateDepthBorder	152.4			6" depth border
costPerHydraulicEjector	4200	Currency	US Dollar	
costPerPorcerax	4800	Currency	US Dollar	
costPerSVGHOTDrop	6000	Currency	US Dollar	
costPerStandardHotDrop	3750	Currency	US Dollar	
costPerUnscrewingMechanism	4200	Currency	US Dollar	
cycleTimeAdjustmentFactor	1			Factor used to globally increase or decrease cycle time evaluations per process.
defaultCostPerLabel	1.00	Currency	US Dollar	Default cost of applying one label to the part. User may override via Process Setup Option
defaultInsertCost	12	Currency	US Dollar	Default cost of one insert. User may override via Process Setup Options.
defaultMoldConstruction	'Standard Mold Base'			'Standard Mold Base' versus 'Custom Mold Base' as the default for the VPE
defaultNumCavities	1			Default value for the number of cavities in one mold. User may override via Process Setup
defaultNumDropBoxes	0			Default number of drop boxes used by the plant. User may override via Process Setup Opti
defaultNumHotDrops	0			
defaultNumLabels	0			Default number of labels applied to the part. User may override via Process Setup Options.
defaultNumTryOuts	3			
defaultNumberOfDrops	1			Default number of drop boxes used by the plant. User may override via Process Setup Opti
defaultPercentGas	0			Default usage percentage by part weight for physical gas. Only applicable for Structural Fc

- 2 To modify a variable's value, click in the **String Value** field and enter the new value.

To modify unit type, double-click in the **Unit Type Name** field and select a unit type from the dropdown menu. This may change the **Unit** field, which you cannot directly modify.

To modify a variable's description, click in the **Notes** field and enter a new value.

- 3 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.



You cannot modify the **Variable Name** field.

Viewing and Modifying VPE Variables

To view and modify VPE variables, follow these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information**, and double click **Cost Model Variables**. The Cost Model Variables table appears in the editing pane.



Variable Name	String Value	Unit Type Name	Unit	Notes
Inherited:				
applyLabelRate	5	Time	second	Rate at which an operator can apply one label.
applyMoldReleaseRate	371612			Rate at which a mold release agent is applied to the mold surface.
betweenPartsMoldBorder	76.2	Length	millimeter	3" between parts in the mold
betweenPartsMoldBorder150T	38.1	Length	millimeter	1.5" between parts in the mold if press tonnage < 150T
betweenPartsMoldBorder300T	50.8	Length	millimeter	2" between parts in the mold if press tonnage < 300T
betweenPartsMoldBorder500T	76.2	Length	millimeter	3" between parts in the mold if press tonnage > 300T
cavityPlateDepthBorder	152.4			make 6" also, same as core border
clampForceSafetyFactor	1.1			multiplier of clamp pressure required to insure that the machine is capable of applying the
clampSpacing	127			Default distance between adjacent clamps along the mold's parting line.
clampingRate	2			Rate at which a clamp can open or close. Determines the open and close mold cycle time.
coarseTextureCostperSqM	15500			Per square meter, from \$10 per mold area square inch to texture the surface
colorantCost	0.75	CostPerMass	US Dollar / kilogram	Cost of colorant per unit mass. The colorant cost is multiplied by the rough mass of the p.
coreInstallRate	15	Time	second	Rate at which an operator can install one core.
corePlateDepthBorder	152.4			6" depth border
costPerHydraulicEjector	4200	Currency	US Dollar	
costPerPorcerax	4800	Currency	US Dollar	
costPerSVGHetDrop	6000	Currency	US Dollar	
costPerStandardHotDrop	3750	Currency	US Dollar	
costPerUnscrewingMechanism	4200	Currency	US Dollar	
cycleTimeAdjustmentFactor	1			Factor used to globally increase or decrease cycle time evaluations per process.
defaultCostPerLabel	1.00	Currency	US Dollar	Default cost of applying one label to the part. User may override via Process Setup Options.
defaultInsertCost	12	Currency	US Dollar	Default cost of one insert. User may override via Process Setup Options.
defaultMoldConstruction	Standard Mold Base			'Standard Mold Base' versus 'Custom Mold Base' as the default for the VPE
defaultNumCavities	1			Default value for the number of cavities in one mold. User may override via Process Setup
defaultNumDropBoxes	0			Default number of drop boxes used by the plant. User may override via Process Setup Opti
defaultNumHotDrops	0			
defaultNumLabels	0			Default number of labels applied to the part. User may override via Process Setup Options.
defaultNumTryOuts	3			
defaultNumberOfDrops	1			Default number of drop boxes used by the plant. User may override via Process Setup Opti
defaultPercentGas	0			Default usage percentage by part weight for physical gas. Only applicable for Structural Fc

- 2 To modify a variable's value, click in the **String Value** field and enter the new value.
To modify unit type, double-click in the **Unit Type Name** field and select a unit type from the dropdown menu. This may change the **Unit** field, which you cannot directly modify.
To modify a variable's description, click in the **Notes** field and enter a new value.
- 3 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

You cannot modify the **Variable Name** field.

Creating New Plant Variables



To create a new plant variable, follow these steps:

- 1 Click in the **Variable Name** field at the bottom of the plant variables table, under **New**;, and enter the name of the new variable.
- 2 In the **String Value** field, enter the new variable's value.
- 3 For variables with values that have an associated unit, click in the **Unit Type Name** field and select a unit type from the dropdown menu. The unit designation appears in the **Unit** field.
- 4 In the Notes field, enter an optional description.
- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Adding New Plant Variables](#) in [Common Task Examples](#) for examples.

Deleting Plant Variables

To remove a plant variable, follow these steps:

- 1 Right click on a row in the plant variables table, and select **Remove** from the context menu.
- 2 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 3 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Lookup Tables

Lookup tables provide tabular data. Global lookup tables provide information that applies to a cost model in general; non-global lookup tables are specific to a process or operation. A lookup table that is operation-specific is specific to an occurrence of an operation in the process-GCD-operation hierarchy, and so has an associated path that names the operation's ancestor operations and process.

Each lookup table has an associated CSL standard input. CSL modules can access a lookup table with a query that uses the table's associated input. Here is an example that queries the lookup table `tubeLaserCutting`, which is associated with the Tube Laser process in the Bar & Tube Fab cost model:

```
cutRateEntry = select first(cr) from tubeLaserCutting cr _
  where materialFamily == cr.materialCutCodeFamily and _
  part.crossSection.thickness <= cr.materialThickness and _
  cr.power == machine.power _
```

```
order by cr.materialThickness asc
```

See [CSL Language Overview](#) for more information on CSL.

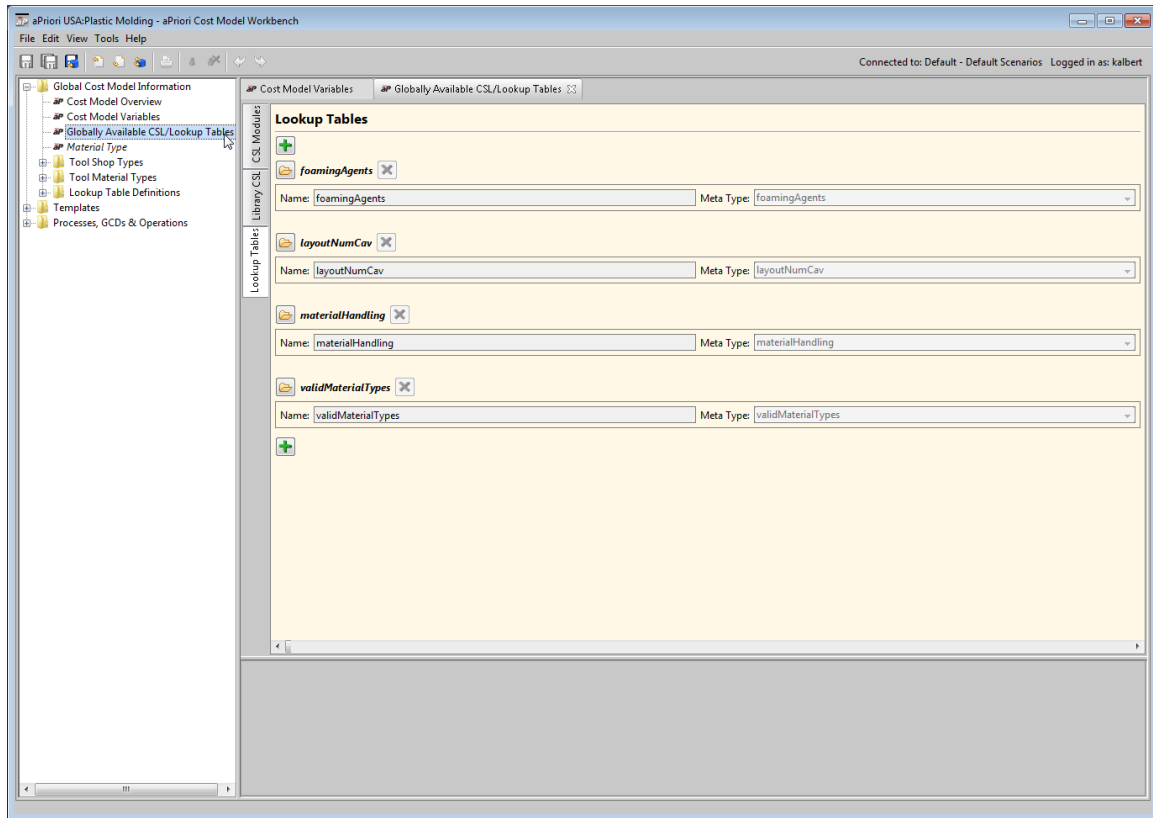
CSL field names of lookup table entries, such as `materialCutCodeFamily` and `materialThickness`, are specified in the **Field Name** field of the table's Lookup Table Definition (see [Viewing and Modifying Lookup Table Definitions](#)). The corresponding attribute name displayed in the VPE Manager is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column grouping, and column properties displayed in the VPE Manager—see [Working with Column Groups and Column Properties](#).

- This section covers the following tasks:
- [Navigating to Lookup Tables](#)
- [Viewing and Modifying Lookup Tables](#)
- [Viewing and Modifying Lookup Table Definitions](#)
- [Creating Lookup Table Definitions](#)
- [Deleting Lookup Table Definitions](#)
- [Adding New Lookup Tables](#)
- [Removing Lookup Tables](#)

Navigating to Lookup Tables

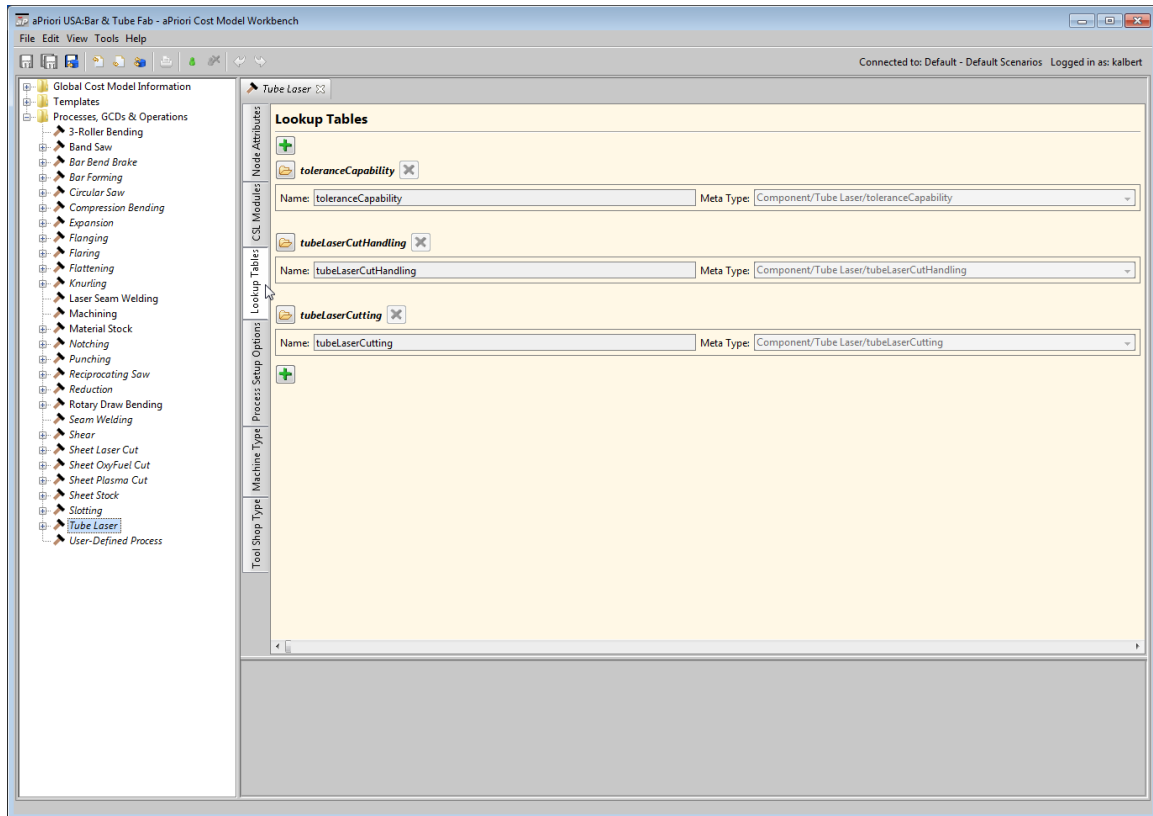
To navigate to the lookup tables that are global to the current process group, follow these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information**, and double click **Globally Available CSL/Lookup Tables**.
- 2 In the editing pane, select the **Lookup Tables** tab. The editing pane displays information about the global lookup tables.



To navigate to the lookup tables for a given process or operation, follow these steps:


- 1 In the CMWB navigation pane, expand **Processes, GCDs & Operations**; double-click the desired process, or navigate to the desired operation and double-click it.
- 2 In the editing pane, select the **Lookup Tables** tab. The editing pane displays information about the specified node's lookup tables.



To navigate to the lookup tables for a given branch node, follow these steps:

- 1 In the CMWB navigation pane, expand **Templates**, navigate to the desired branch node, and double-click it.
- 2 In the editing pane, select the **Lookup Tables** tab. The editing pane displays information about the specified node's lookup tables.

For each lookup table, the editing pane displays the following:


- Table name
- Table meta type, the pathname to the table's schema definition (see [Viewing and Modifying Lookup Table Definitions](#)).
- Folder icon, , for table viewing or editing
- X symbol for module deletion

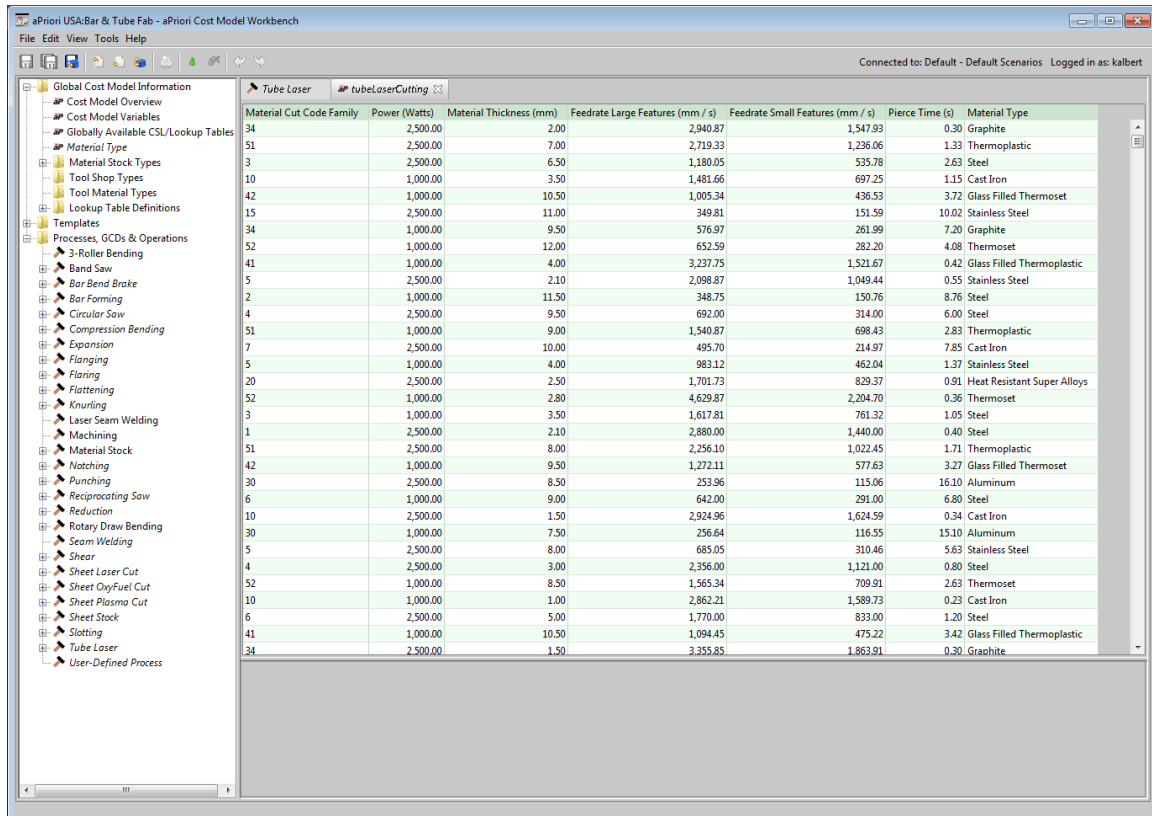
The editing pane also displays a + icon for adding a table. See [Adding New Lookup Tables](#).

Viewing and Modifying Lookup Tables

To view and modify a lookup table, follow these steps:




- 1 Navigate to the lookup tables for the desired lookup table's associated node (see [Navigating to Lookup Tables](#)).

- Click  next to the name of the table you want to modify. The lookup table appears in the editing pane.



The screenshot shows the 'aPriori USA Bar & Tube Fab - aPriori Cost Model Workbench' application. The left sidebar contains a tree view of the cost model structure, with 'Tube Laser' selected under 'Processes, GCDs & Operations'. The main window displays a table with the following columns: Material Cut, Code Family, Power (Watts), Material Thickness (mm), Feedrate Large Features (mm / s), Feedrate Small Features (mm / s), Pierce Time (s), and Material Type. The table contains 34 rows of data.

Material Cut	Code Family	Power (Watts)	Material Thickness (mm)	Feedrate Large Features (mm / s)	Feedrate Small Features (mm / s)	Pierce Time (s)	Material Type
34		2,500.00	2.00	2,940.87	1,547.93	0.30	Graphite
51		2,500.00	7.00	2,719.33	1,236.06	1.33	Thermoplastic
3		2,500.00	6.50	1,180.05	535.78	2.63	Steel
10		1,000.00	3.50	1,481.66	697.25	1.15	Cast Iron
42		1,000.00	10.50	1,005.34	436.53	3.72	Glass Filled Thermoset
15		2,500.00	11.00	349.81	151.59	10.02	Stainless Steel
34		1,000.00	9.50	576.97	261.99	7.20	Graphite
52		1,000.00	12.00	652.59	282.20	4.08	Thermoset
41		1,000.00	4.00	3,237.75	1,521.67	0.42	Glass Filled Thermoplastic
5		2,500.00	2.10	2,098.87	1,049.44	0.55	Stainless Steel
2		1,000.00	11.50	348.75	150.76	8.76	Steel
4		2,500.00	9.50	692.00	314.00	6.00	Steel
51		1,000.00	9.00	1,540.87	698.43	2.83	Thermoplastic
7		2,500.00	10.00	495.70	214.97	7.85	Cast Iron
5		1,000.00	4.00	983.12	462.04	1.37	Stainless Steel
20		2,500.00	2.50	1,701.73	829.37	0.91	Heat Resistant Super Alloys
52		1,000.00	2.80	4,629.87	2,204.70	0.36	Thermoset
3		1,000.00	3.50	1,617.81	761.32	1.05	Steel
1		2,500.00	2.10	2,880.00	1,440.00	0.40	Steel
51		2,500.00	8.00	2,256.10	1,022.45	1.71	Thermoplastic
42		1,000.00	9.50	1,272.11	577.63	3.27	Glass Filled Thermoset
30		2,500.00	8.50	253.96	115.06	16.10	Aluminum
6		1,000.00	9.00	642.00	291.00	6.80	Steel
10		2,500.00	1.50	2,924.96	1,624.59	0.34	Cast Iron
30		1,000.00	7.50	256.64	116.55	15.10	Aluminum
5		2,500.00	8.00	685.05	310.46	5.63	Stainless Steel
4		2,500.00	3.00	2,356.00	1,121.00	0.80	Steel
52		1,000.00	8.50	1,565.34	709.91	2.63	Thermoset
10		1,000.00	1.00	2,862.21	1,589.73	0.23	Cast Iron
6		2,500.00	5.00	1,770.00	833.00	1.20	Steel
41		1,000.00	10.50	1,094.45	475.22	3.42	Glass Filled Thermoplastic
34		2,500.00	1.50	3,355.85	1,863.91	0.30	Graphite

- Select **Override Object** from the CMWB Edit menu, or click the override icon, , in the toolbar.
- You can modify field values, and add and remove table rows.
To modify a field value, click in the field, enter the new value, and press return.
To add a table row, click in a field in the (empty) last row of the lookup table, and enter a value. Enter values in the other fields of the same row.
To remove a row, right-click on it and select **Remove**.
- Select **Save** from the File menu, or click  in the toolbar, to save your changes.
- To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the File menu, or click  in the toolbar.

Viewing and Modifying Lookup Table Definitions

Multiple lookup tables can use the same lookup table definition. A lookup table definition specifies lookup table metadata. In particular, it specifies how many columns are in any table that uses the definition; and it specifies information about each column, such as the column's name, default value, and units.




Follow these steps to navigate to a lookup table definition:

- 1 In the CMWB navigation pane, expand Global Cost Model Information; under that, expand **Lookup Table Definitions**.
- 2 Double click the desired lookup table definition. The definition table appears in the editing pane.

The definition table has one row for each column of any table that uses the definition. Each row of the definition has the following fields:

- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**



Follow these steps to modify a lookup table definition:

- 1 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 2 You can modify field values, and add and remove table rows.
To modify a field value, click in the field, and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu.
To add a table row, click in the **Field Name** field in the (empty) last row of the definition table, and enter a name. Enter values in the other fields of the same row. Manage the display of new attributes as described in [Working with Column Groups and Column Properties](#).
To remove a row, right-click on it and select **Remove**.
- 3 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Creating Lookup Table Definitions

When you create a new lookup table, you specify the definition that the new table should use (which specifies information--such as name and default value--about each column that the new table will have). If you want to create a table that does not conform to any existing definition, you must create a new lookup table definition.

Follow these steps to create a new lookup table definition:

- 1 Right-click any node in the CMWB navigation pane, and select **New > Lookup Table Definition....** The **New Lookup Table Definition** dialog appears.
- 2 Enter a name for the new definition in the dialog's **Name** field.
- 3 Optionally, select an existing definition from the dialog's **Copy Definition From** field. This allows you to edit a copy of an existing definition in order to create the new one.
- 4 Click **OK**. The new definition appears in the editing pane.
- 5 Modify the definition table as needed (see [Viewing and Modifying Lookup Table Definitions](#)).
- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Adding Lookup Tables](#) in Common Task Examples for an example.

You can also create a new definition in the course of creating a new lookup table. See [Adding New Lookup Tables](#).



Deleting Lookup Table Definitions

To delete a lookup table definition, follow these steps:





- 1 In the CMWB navigation pane, expand **Global Cost Model Information**; under that, expand **Lookup Table Definitions**.
- 2 Right click the desired lookup table definition, and select **Delete Object** from the context menu.

Adding New Lookup Tables

Follow these steps to add a new lookup table:





- 1 Navigate to global cost model data (see [Navigating to Global Data](#)) or to the data for the node with which you want to associate the new lookup table (see [Navigating to the Data for a Process, Operation or Branch Node](#)).
- 2 Click the **Lookup Tables** tab in the editing pane.
- 3 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 4 Click  in the editing pane. Information fields for the new lookup table are added to the editing pane.
- 5 Enter the name of the new lookup table in the **Name** field.

- 6 Click the **Meta Type** field, and select the definition that the new lookup table should conform to.

If you need a new definition, scroll to the bottom of the dropdown menu and select **Define New Type...**; the Define New Lookup Table Type dialog appears. Follow the steps in [Creating Lookup Table Definitions](#), beginning with step 2.
 - 7 Select **Save** from the **File** menu or click  in the toolbar. The information display for the new table may change location within the editing pane.
 - 8 Click  next to the name of the lookup table. An empty table (with the columns specified by the definition) appears in the editing pane.
 - 9 Add data to the table.
 - 10 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
 - 11 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.
- See [Adding Lookup Tables](#) in Common Task Examples for an example.

Removing Lookup Tables

To delete a lookup table, follow these steps:

- 1 Navigate to global cost model data (see [Navigating to Global Data](#)) or to the data for the node with the lookup table that you want to delete (see [Navigating to the Data for a Process, Operation or Branch Node](#)).
- 2 Click the **Lookup Tables** tab in the editing pane.
- 3 Select **Override Object** from the **CMWB Edit** menu, or click the override icon, , in the toolbar.
- 4 Click  next to the lookup table that you want to remove.
- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Machine Metadata

Every process has an associated set of machines. By using the VPE manager, you can add and remove machines, as well as view and modify the attributes of a process's machines (see the [VPE Administration Guide](#)). Every process also has an associated machine type, which specifies machine metadata. It specifies what type of information is

stored about the associated machines, including the names and default values of machine attributes. You manage machine metadata with the CMWB.

CSL modules can access machine attributes with the CSL standard input `machine`. This input has one field for each machine attribute. Here is an example from the Tube Laser process in the Bar & Tube Fab process group. It accesses the `tubeAxialBarFeedRate` attribute of the current machine:

```
holeRapidTraverseTime = _  
    (averageDistanceBetweenHoles / machine.tubeAxialBarFeedRate)
```

See [CSL Language Overview](#) for more information on CSL.

CSL field names, such as `tubeAxialBarFeedRate`, are specified in the **Field Name** field of the Machine Type table (see [Viewing and Modifying Machine Types](#)). The corresponding attribute name displayed in the VPE Manager and aPriori's **Edit Machine Selection** dialog is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column grouping, and column properties displayed to the end user—see [Working with Column Groups and Column Properties](#).

This section covers the following tasks:

- [Viewing and Modifying Machine Types](#)
- [Adding Machine Types](#)
- [Deleting Machine Types](#)
- [Adding or Modifying a Machine Field for Multiple Processes](#)

Viewing and Modifying Machine Types


To view and modify the machine type for a given process, follow these steps:

- 1 Expand **Processes, GCDs & Operations**, and double-click the desired process.
- 2 In the editing pane, select the **Machine Type** tab. The editing pane displays the machine type table.

Field Name	Description	Default Value Text	Unit Type Name	Unit	Property Type Name	Notes
averageUtilization		0.95			double	
clampForce		55000.0	kN	kiloNewton	double	
description					string	
dryCycleTime		12.5	Time	second	double	
injectionRate		3710000.0	FlowRate	millimeter^3 / second	double	
laborRatio		0.5			double	
laborTimeStandard		1.0			double	
maxMoldHeight		2500.0	Length	millimeter	double	
minMoldHeight		1500.0	Length	millimeter	double	
moldEfficiency		0.95			double	
numOperators	Number of operators tending the machine. Can be less than 1.				double	
periodOverheadCoefficient		0.0			double	
setupTime		2.0	hr	hour	double	
shotSize		72000.0		g	double	
tieBarDistanceH		2500.0	Length	millimeter	double	
tieBarDistanceV		2500.0	Length	millimeter	double	
workCenterLaborRate		15.2	Currency/hr	US Dollar / hour	double	
workCenterOverheadMultiplier		0.0			double	
workCenterOverheadRate		224.98	Currency/hr	US Dollar / hour	double	

The machine type table has one row for each column of the current process's machine table (that is, the type table has one row for each machine attribute). Each row of the type table has the following fields:

- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**



3 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.

4 You can modify field values, and add and remove table rows.

To modify a field value, click in the field and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu. Modifiable fields contain the pencil icon.

To add a table row, click in the **Field Name** field in the (empty) last row of the type table, and enter a name. Enter values in the other fields of the same row. Manage the display of new machine attributes as described in [Working with Column Groups and Column Properties](#).





To remove a row, right-click on it and select **Remove**.

- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Defining and Modifying Machine Types](#) in Common Task Examples for an example.

Adding Machine Types


If a process has no associated machine type, you can add one by following these steps:



- 1 Expand **Processes, GCDs & Operations**, and double-click the desired process.
- 2 In the editing pane, select the **Machine Type** tab. The editing pane displays the machine type page with no table.
- 3 Click . An empty machine type table appears in the editing pane.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes. The table is populated with the required fields machine tables:
 - o averageUtilization
 - o goodPartYield
 - o laborTimeStandard
 - o periodOverheadCoefficient
 - o setupTime
 - o workCenterLaborRate
 - o workCenterOverheadMultiplier
 - o workCenterOverheadRate
- 5 Enter values into the fields. As soon as a value is recorded in a row, a new blank row appears below it. See [Viewing and Modifying Machine Types](#) for more information.
- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Defining and Modifying Machine Types](#) in Common Task Examples for an example.

Deleting Machine Types

Follow these steps to delete a machine table

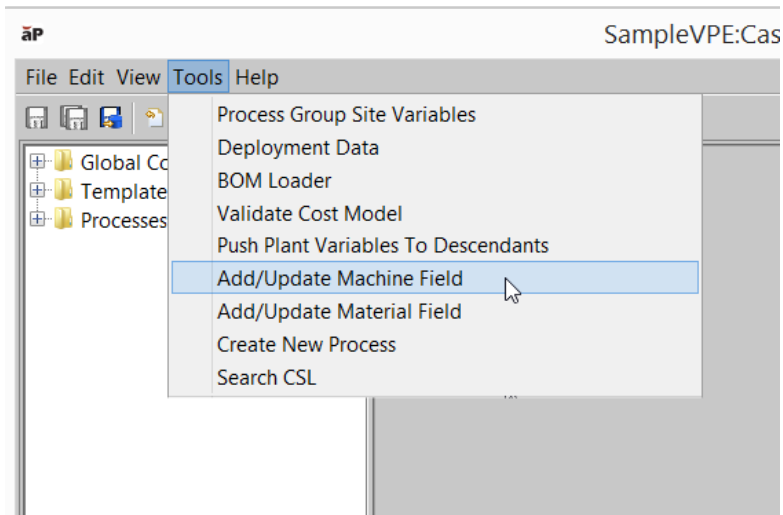
- 1 Expand **Processes, GCDs & Operations**, and double-click the desired process.
- 2 In the editing pane, select the **Machine Type** tab. The editing pane displays the machine type page with no table.
- 3 Click . The table is removed from the editing pane.

- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding or Modifying a Machine Field for Multiple Processes

To add or modify a machine field for multiple processes at once, follow these steps (you can also perform these steps from the VPE Manager, and a cost model need not be open):

- 1 Select **Add/Update Machine Field** from the Tools menu.



The **Add/Update Machine Field** dialog appears:

Key usage notes

- Use this feature to add or update machine columns to multiple processes within a given cost model in a VPE.
- The data used to populate those columns must be entered in the VPE Manager.
- If the field already exists, then it will be updated with the definition provided below.

VPE name:

Process Group:

Source machine table:

Process Name

Field Name (required):

Display Name:

Description:

Property Type Name:

Default Value Text:

Unit Type Name:

Notes:

Parent Group:

OK Cancel

2 Fill in the following dialog fields:

- **VPE name:** VPE that contains the processes whose machine types you want to modify. Setting this field populates the dropdown list of choices for the **Process Group** field.
- **Process Group:** process group that contains the processes whose machine types you want to modify. Setting this field populates the dropdown list of choices for the **Source machine table** field.
- **Source machine table:** processes whose machine types you want to modify. Click to select the first process; control click to select each subsequent process. Shift click to mark the end of a range to be selected. Use Ctrl-A to select all processes. Selected processes that do not define any machine type (such as User-Defined Process) will be unaffected by the addition or modification of the field; such processes will continue to lack any associated machine type.
- **Field Name:** name of the machine field that you want to add or modify.

- **Display Name:** optional name of the field as displayed to the end user. If you don't supply one, the display name is the result of converting the **Field Name** to space-separated, initial-cap words.
- **Description:** optional description of the field you want to add or modify.
- **Property Type Name:** CSL data type (such as **double**, **int**, or **string**) used for the field value.
- **Default Value Text:** optional default value of the field you want to add or modify.
- **Unit Type Name:** optional unit type (such as **Mass**, **Length**, or **Time**) used for the field value. This name must be a case-sensitive, exact match for an aPiori-supported unit type.
- **Notes:** optional annotation.
- **Parent Group:** Optional column group for the new or modified machine field. See Working with Column Groups and Column Properties. If you don't supply a group, the new machine field's column group is **Other**.

Key usage notes

- Use this feature to add or update machine columns to multiple processes within a given cost model in a VPE.
- The data used to populate those columns must be entered in the VPE Manager.
- If the field already exists, then it will be updated with the definition provided below.

VPE name: SampleVPE

Process Group: Casting

Source machine table:

Process Name
No Bake
No Cost Feature
Oil Core
PM Cleaning
PM CoreMaking

Field Name (required): changeTime

Display Name: Change Time

Description: Time to change corebox

Property Type Name: double

Default Value Text: 0.0

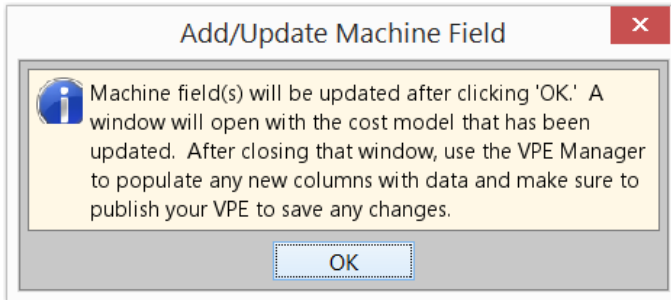
Unit Type Name: Time

Notes:

Parent Group: Time


OK Cancel

- 3 Click **OK** to override the machine types and make the specified changes. A notification dialog appears:



- 4 Click **OK** in the notification dialog. aPriori opens the modified cost model in the CMWB, if it's not already open.

Note: your changes are not saved to your private workspace until you publish the changes to the public cost model (see step 6, below). That is, unless you publish the changes, they will not persist across invocations of aPriori.

- 5 Use the VPE Manager to add or modify values for the new or modified machine fields.
- 6 To save your changes to your private workspace and incorporate the changes into the public cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Material Metadata

Every process group has an associated set of materials. By using the VPE Manager, you can add and remove materials, as well as view and modify the attributes of a process group's materials (see the VPE Administration Guide). Every process group also has an associated material type, which specifies material metadata. It specifies what type of information is stored about the process group's materials, including the names and default values of material attributes. You manage material metadata with the CMWB.

CSL modules can access material attributes with the CSL standard input `material`. This input has one field for each material attribute. Here is an example from the Injection Molding process in the Plastic Molding process group. It accesses the `canIM_SFM` attribute of the current material:

```
Rule CompatibleMaterial: material.canIM_SFM
Message CompatibleMaterial: _
    'Failed because you cannot Injection Mold this type of material'
```

See [CSL Language Overview](#) for more information on CSL.

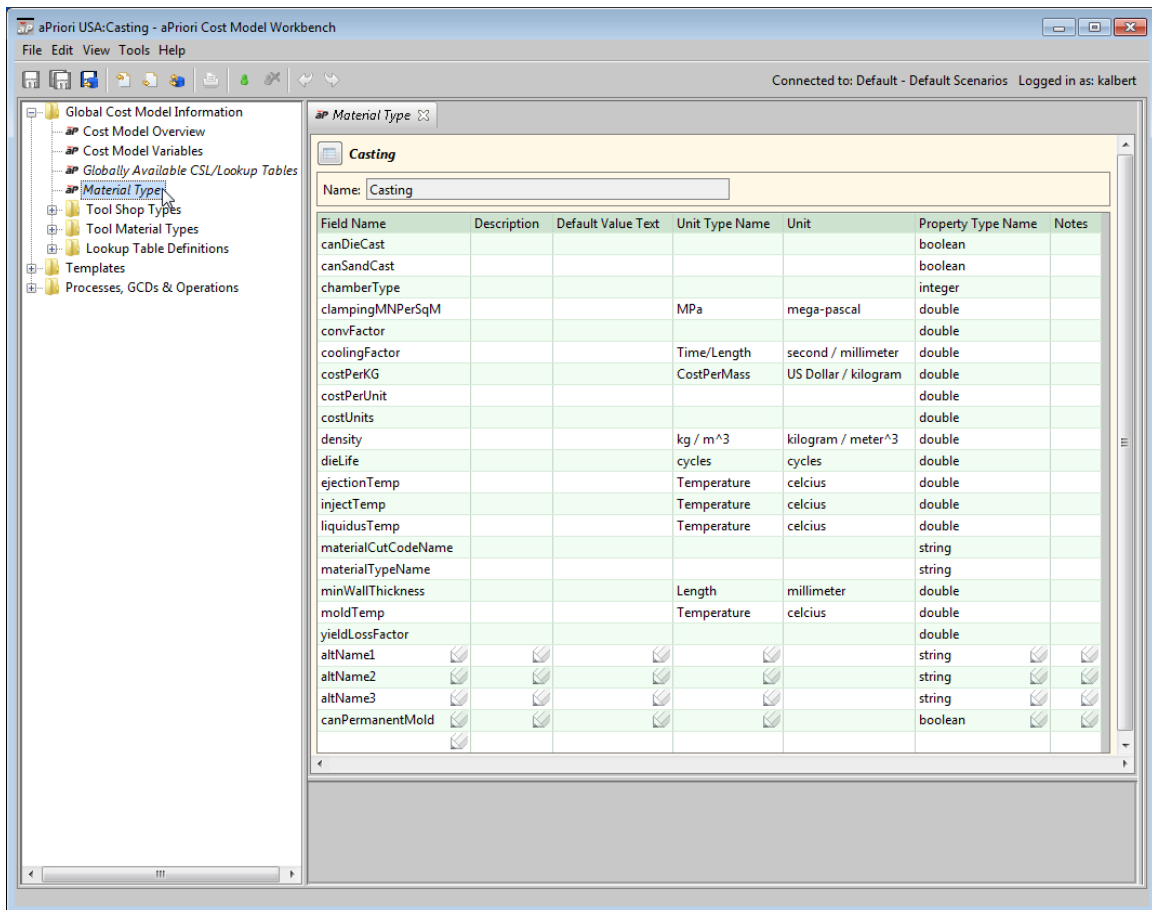
CSL field names, such as `canIM_SFM`, are specified in the **Field Name** field of the Material Type table (see [Viewing and Modifying Material Types](#)). The corresponding attribute name displayed in the VPE Manager and the aPriori **Material Selection** dialog is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column

grouping, and column properties displayed to the end user—see [Working with Column Groups and Column Properties](#).

Viewing and Modifying Material Types




Follow these steps to view and modify material types:

- 1 Expand **Global Cost Model Information**, and double-click **Material Type**. The editing pane displays the material type table.



The material type table has one row for each column of the current process group's material table (that is, the type table has one row for each material attribute). Each row of the type table has the following fields:

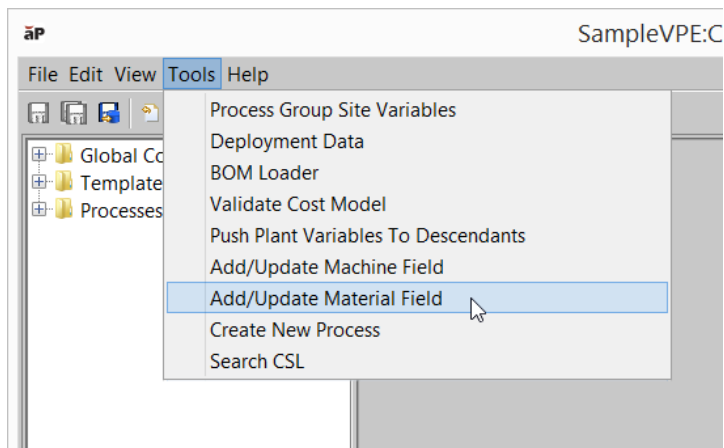
- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**

- 2 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 3 You can modify field values, and add and remove table rows.
To modify a field value, click in the field and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu. Modifiable fields contain the pencil icon.
To add a table row, click in the **Field Name** field in the (empty) last row of the type table, and enter a name. Enter values in the other fields of the same row. Manage the display of new material attributes as described in [Working with Column Groups and Column Properties](#).
To remove a row, right-click on it and select **Remove**.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding or Modifying a Material Field for Multiple Process Groups

To add or modify a material field for multiple process groups at once, follow these steps (you can also perform these steps from the VPE Manager, and a cost model need not be open.):

- 1 Select **Add/Update Material Field** from the Tools menu.



The **Add/Update Machine Field** dialog appears:

Key usage notes

- Use this feature to add or update material columns to multiple cost models in a VPE.
- The data used to populate those columns must be entered in the VPE Manager.
- If no parent group is provided for the field then it will be added to the last group in the table.
- If the field already exists, then it will be updated with the definition provided below.

Select VPE:

Select Process Group(s):

Process Group Name

Field Name (required):

Display Name:

Description:

Default Value Text:

Unit Type Name:

Property Type Name:

Notes:

Parent Group:

OK Cancel

2 Fill in the following dialog fields:

- **Select VPE:** VPE that contains the process groups whose material types you want to modify. Setting this field populates the dropdown list of choices for the **Select Process Group(s)** field.
- **Select Process Group(s):** process groups whose material types you want to modify. Click to select the first process group; control click to select each subsequent process group. Shift click to mark the end of a range to be selected. Use Ctrl-A to select all process groups. Selected process groups that do not define any material type (such as secondary process groups) will be unaffected by the addition or modification of the field; such process groups will continue to lack any associated material type.
- **Field Name:** name of the material field that you want to add or modify.
- **Display Name:** optional name of the field as displayed to the end user. If you don't supply one, the display name is the result of converting the **Field Name** to space-separated, initial-cap words.

- **Description:** optional description of the field you want to add or modify.
- **Default Value Text:** optional default value of the field you want to add or modify.
- **Unit Type Name:** optional unit type (such as **Mass**, **Length**, or **Time**) used for the field value. This name must be a case-sensitive, exact match for an aPriori-supported unit type.
- **Property Type Name:** CSL data type (such as **double**, **int**, or **string**) used for the field value.
- **Notes:** optional annotation.
- **Parent Group:** Optional column group for the new or modified machine field. See Working with Column Groups and Column Properties. If you don't supply a group, the new machine field's column group is **Other**.

Key usage notes

- Use this feature to add or update material columns to multiple cost models in a VPE.
- The data used to populate those columns must be entered in the VPE Manager.
- If no parent group is provided for the field then it will be added to the last group in the table.
- If the field already exists, then it will be updated with the definition provided below.

Select VPE: SampleVPE

Select Process Group(s):

- Process Group Name
- Bar & Tube Fab
- Casting
- Forging
- Heat Treatment
- Machining

Field Name (required): altName6

Display Name: XYZ Name

Description:

Default Value Text:

Unit Type Name:

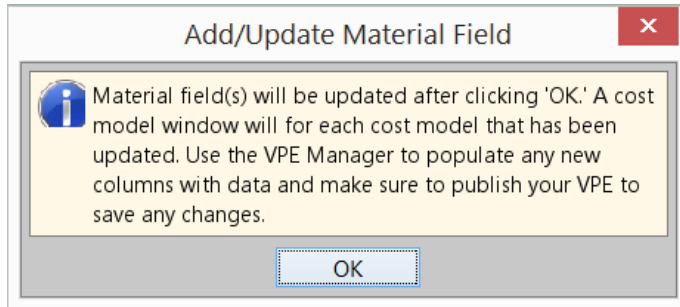
Property Type Name: double


Notes:

Parent Group:

OK Cancel

- 3 Click **OK** to override the material types and make the specified changes. A notification dialog appears:



- 4 Click **OK** in the notification dialog. aPriori opens the modified cost models in the CMWB, if they're not already open.
Note: your changes are not saved to your private workspace until you publish the changes to the public cost model (see step 6, below). That is, unless you publish the changes, they will not persist across invocations of aPriori.
- 5 Use the VPE Manager to add or modify values for the new or modified material fields.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Material Stock Metadata

Some process groups have an associated set of material stocks. By using the VPE Manager, you can add and remove material stocks, as well as view and modify the attributes of a process group's material stocks (see the VPE Administration Guide). If a process group has material stocks, it also has an associated set of material stock types, which specify material stock metadata. Each material stock is an instance of a material stock type. Multiple material stocks can be instances of the same material stock type. Each material stock type specifies what kind of information is stored about its instances, including the names and default values of its instances' attributes. You manage material stock metadata with the CMWB.

CSL modules can access material stock attributes with the CSL standard input `stock`. This input has one field for each material stock attribute. Here is an example from the Tube Laser process in the Bar & Tue Fab process group. It accesses the `stockForm` attribute of the current material stock:

```
Rule IncompatibleStockForm1: _
    not (stock.stockForm == 'Round Bar' or stock.stockForm ==
        'ROUND_BAR')
Message IncompatibleStockForm1: 'Tube laser cannot cut round solid
bar stock'
```

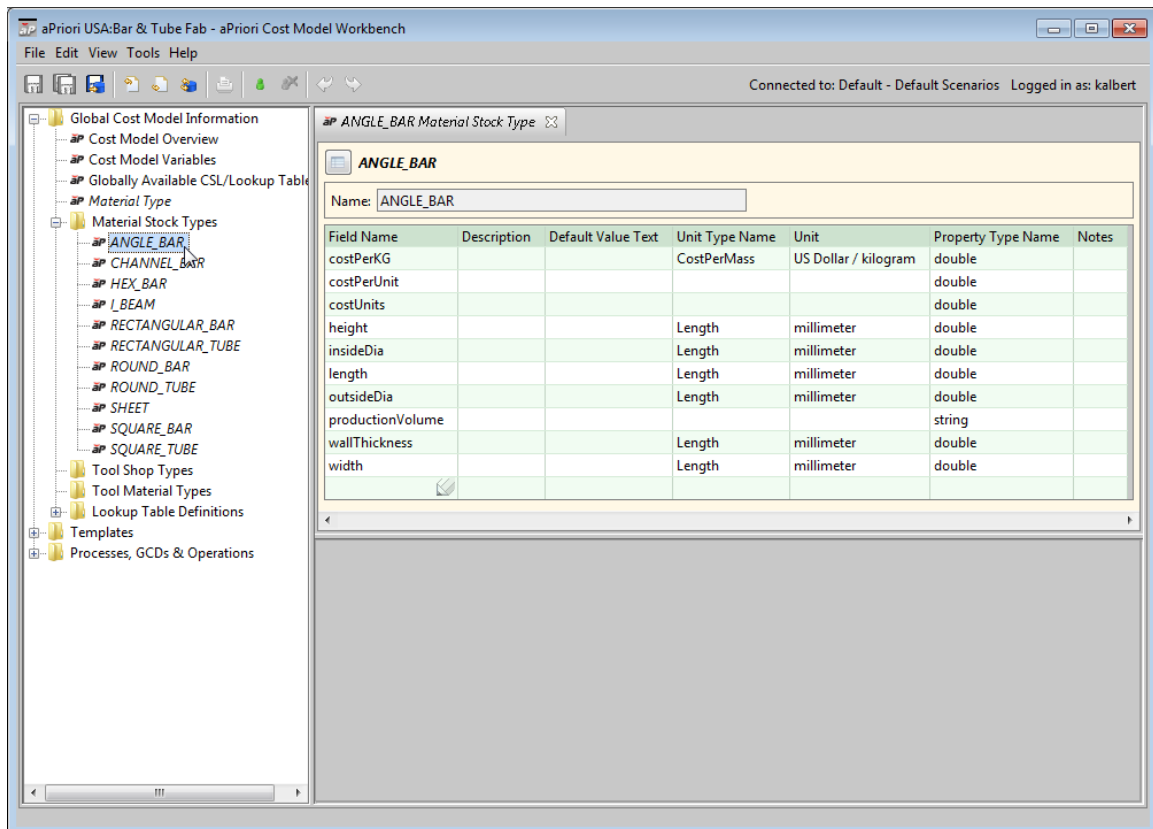
See [CSL Language Overview](#) for more information on CSL.

CSL field names, such as `stockForm`, are specified in the **Field Name** field of the Material Stock Type tables (see Viewing and Modifying Material Stock Types). The corresponding attribute name displayed in the VPE Manager and the aPriori **Material Selection** dialog is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column grouping, and column properties displayed to the end user—see Working with Column Groups and Column Properties.

Viewing and Modifying Material Stock Types




Follow these steps to view and modify material types:

- 1 Expand **Global Cost Model Information**, and double-click **Material Stock Types**. The editing pane displays the material stock type table.



The material stock type table has one row for each column of the current process group's material stock table (that is, the type table has one row for each stock attribute). Each row of the type table has the following fields:

- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**

- 2 Select **Override Object** from the **CMWB Edit** menu, or click the override icon, , in the toolbar.
- 3 You can modify field values, and add and remove table rows.
 To modify a field value, click in the field and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu. Modifiable fields contain the pencil icon.
 To add a table row, click in the **Field Name** field in the (empty) last row of the type table, and enter a name. Enter values in the other fields of the same row. Manage the display of new material stock attributes as described in [Working with Column Groups and Column Properties](#).
 To remove a row, right-click on it and select **Remove**.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Tool Shop Metadata

A process that uses tooling, such as molds, cores, and actions, can have an associated tool shop, which is a set of variables. By using the VPE Manager, you can add tool shops, and view and modify tool shop variables (see the VPE Administration Guide). Each tool shop is an instance of a tool shop type, which specifies tool shop metadata. It specifies what kind of information is stored for the tool shop, including variable names and default values. You manage tool shop metadata with the CMWB.

CSL modules can access tool shop variables with the CSL standard input `toolshop`. This input has one field for each tool shop variable. Here is an example from the Injection Molding process in the Plastic Molding process group. It accesses the `tryoutRate` variable of the current process's tool shop:

```
IM_Tryouts = tryoutTime * toolShop.tryoutRate *
standCustomMultiplier
```

See [CSL Language Overview](#) for more information on CSL.

CSL field names, such as `tryoutRate`, are specified in the **Field Name** field of the tool shop type table (see [Viewing and Modifying Tool Shop Types](#)). The corresponding variable name displayed in the VPE Manager is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column grouping, and column properties displayed to the end user—see [Working with Column Groups and Column Properties](#).

This section covers the following tasks:

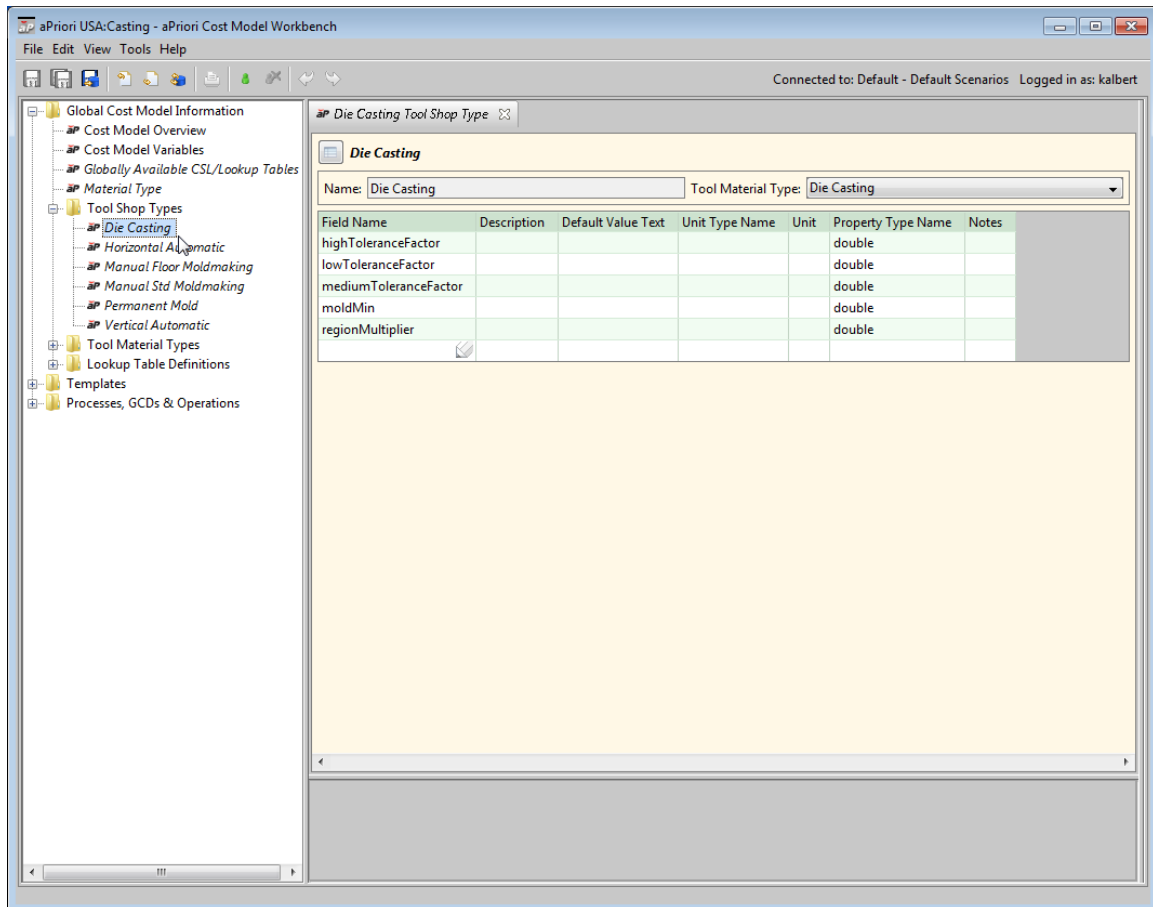
- [Viewing and Modifying Tool Shop Types](#)
- [Adding Tool Shop Types](#)

- Deleting Tool Shop Types
- Associating a Tool Shop Type with a Process
- Removing a Tool Shop Type from a Process

Viewing and Modifying Tool Shop Types




To view and modify a tool shop type, follow these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information**, expand **Tool Shop Types**, and double-click the desired tool shop type. The tool shop type table appears in the editing pane.



The tool shop type table has one row for each tool shop variable. Each row of the type table has the following fields:

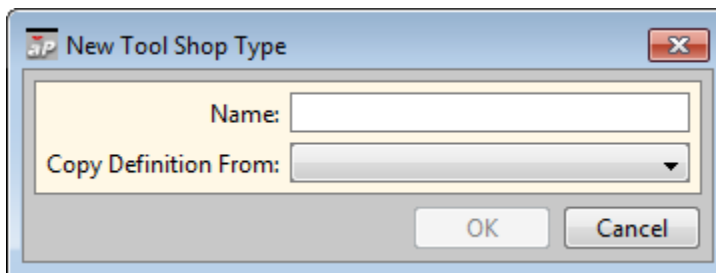
- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**



- 2 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 3 You can modify field values, and add and remove table rows.
To modify a field value, click in the field and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu. Modifiable fields contain the pencil icon.
To add a table row, click in the **Field Name** field in the (empty) last row of the type table, and enter a name. Enter values in the other fields of the same row. Manage the display of new tool shop variables as described in [Working with Column Groups and Column Properties](#).
To remove a row, right-click on it and select **Remove**.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding Tool Shop Types

To add a new tool shop type to a process group, follow these steps:

- 1 In the CMWB navigation pane, right click on any node and select **New > Tool Shop Type...** The **New Tool Shop Type** dialog appears.



- 2 Enter a name for the new tool shop type; optionally, select a tool shop type to copy from.
- 3 Click **OK**. a blank tool shop type table appears in the editing pane.
- 4 Enter values into the fields. As soon as a value is recorded in a row, a new blank row appears below it. See [Viewing and Modifying Tool Shop Types](#) for more information.
- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Deleting Tool Shop Types




If a given tool shop type is not associated with any process and no tool shop is an instance of it, you can delete the tool shop type by following these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information** and then expand **Tool Shop Types**.
- 2 Right click the desired tool shop type, and select **Delete Object** from the context menu.

Associating a Tool Shop Type with a Process




To add a tool shop to a process that does not have one, you must first associate a tool shop type with the process. Once a process has an associated too shop type, you can use VPE Manager to create a tool shop and tool materials for the process.

If a process has no associated tool shop type, you can add one by following these steps:

- 1 In CMWB navigation pane, expand **Processes, GCDs & Operations**, and double-click the desired process.
- 2 In the editing pane, select the **Tool Shop Type** tab. The editing pane displays an empty tool shop type page.
- 3 Click . The tool shop type appears in the editing pane.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Removing a Tool Shop Type from a Process

To disassociate a tool shop type from a process (see [Associating a Tool Shop Type with a Process](#)) follow these steps:

- 1 In CMWB navigation pane, expand **Processes, GCDs & Operations**, and double-click the desired process.
- 2 In the editing pane, select the **Tool Shop Type** tab.
- 3 Click . The tool shop type is removed from the editing pane.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Working with Tool Material Metadata

A process that has an associated tool shop also has a set of associated tool materials. By using the VPE Manager, you can add and remove tool materials, and view and modify tool material attributes (see the VPE Administration Guide). Each of a tool shop's associated materials is an instance of the tool shop type's associated tool material type, which specifies tool material metadata. It specifies what kind of information is stored for the tool materials, including material attribute names and default values. You manage tool material metadata with the CMWB.

CSL modules can access the collection of a process's associated tool materials with the `toolMaterials` field of the CSL standard input `toolshop`. Here is an example from the Injection Molding process in the Plastic Molding process group. It accesses the tool shop's associated material (there is only one in this case), and assigns it to `moldMaterialStandardMoldBase`:

```
moldMaterialStandardMoldBase = _
    select first(m) from toolShop.toolMaterials m //only one base
    material
```

Each element of the value of `toolshop.toolMaterials` has one field for each attribute defined by the tool material type. Here is another example from the Injection Molding process in the Plastic Molding process group. It accesses the tool material attributes `heightMultiplier`, `coeffA`, `coeffB`, and `coeffC`. Note that, as in the example above, the tool material has been assigned to `moldMaterialStandardMoldBase`.

```
heightMultiplier = moldMaterialStandardMoldBase.heightMultiplier

standardMoldBaseCost = _
    (( moldMaterialStandardMoldBase.coeffA * moldAreaInch^2 ) + _
    ( moldMaterialStandardMoldBase.coeffB * moldAreaInch ) + _
    ( moldMaterialStandardMoldBase.coeffC ) + _
    part.boxHeight * heightMultiplier) * _
    standardMoldBaseMaterialMultiplier * regionMultiplier * _
    standardMoldBaseEjectorBoxMultiplier
```

See [CSL Language Overview](#) for more information on CSL.

CSL field names, such as `heightMultiplier`, are specified in the **Field Name** field of the tool material type table (see [Viewing and Modifying Tool Material Types](#)). The corresponding attribute name displayed in the VPE Manager is controlled by the CMWB column properties dialog. This dialog also controls the formatting, column grouping, and column properties displayed to the end user—see [Working with Column Groups and Column Properties](#).

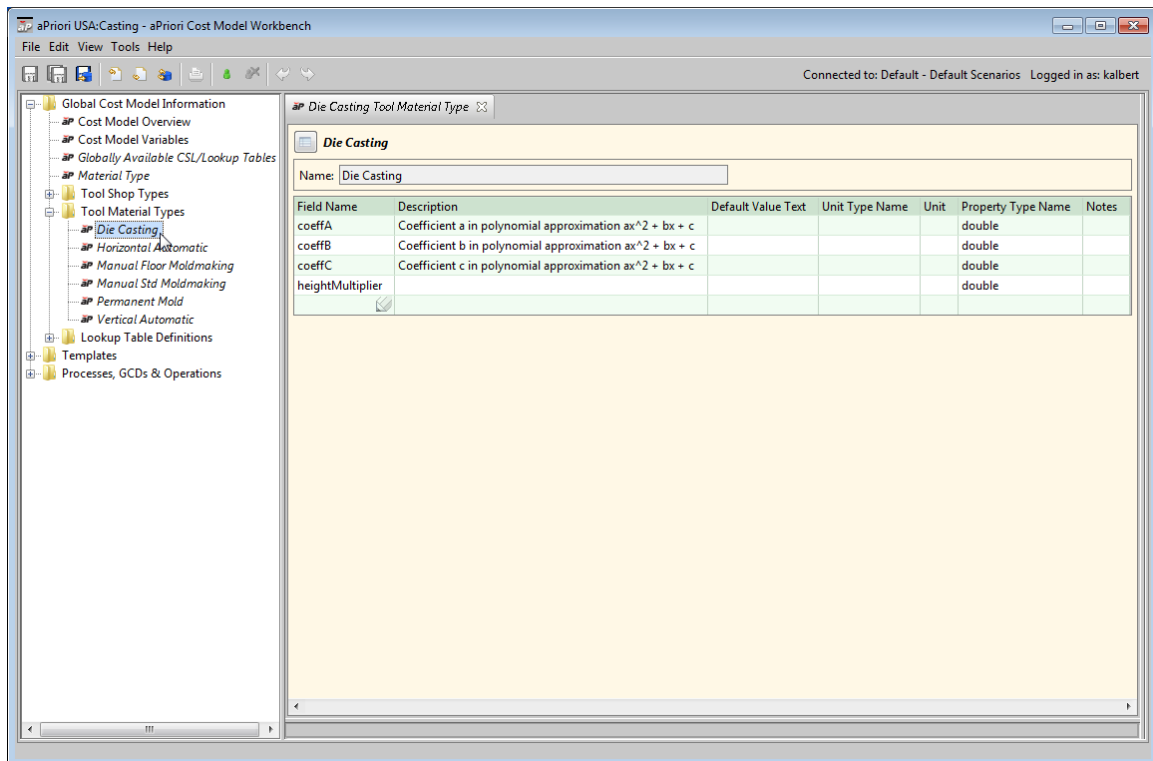
This section covers the following tasks:

- Viewing and Modifying Tool Material Types
- Adding Tool Material Types
- Deleting Tool Material Types

Viewing and Modifying Tool Material Types

To view and modify a tool material type, follow these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information**, expand **Tool Material Types**, and double-click the desired tool material type. The tool material type table appears in the editing pane.



The tool material type table has one row for each column of the current process's tool material table (that is, the type table has one row for each tool material attribute). Each row of the type table has the following fields:

- **Field Name**
- **Description**
- **Default Value Text**
- **Unit Type Name**
- **Unit**
- **Property Type Name**
- **Notes**



- 2 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.

- 3 You can modify field values, and add and remove table rows.

To modify a field value, click in the field and enter the new value, or (for the **Unit Type Name** field) double click in the field and select an item from the dropdown menu. Modifiable fields contain the pencil icon.

To add a table row, click in the **Field Name** field in the (empty) last row of the type table, and enter a name. Enter values in the other fields of the same row. Manage the display of new tool material attributes as described in [Working with Column Groups and Column Properties](#).

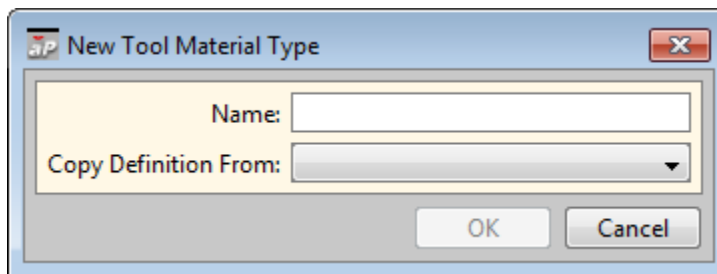
To remove a row, right-click on it and select **Remove**.



- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding Tool Material Types

To add a new tool material type to a process group, follow these steps:

- 1 In the CMWB navigation pane, right click on any node and select **New > Tool Material Type....** The **New Tool Material Type** dialog appears.



- 2 Enter a name for the new tool material type; optionally, select a tool material type to copy from.
- 3 Click **OK**. a blank tool material type table appears in the editing pane.
- 4 Enter values into the fields. As soon as a value is recorded in a row, a new blank row appears below it. See [Viewing and Modifying Tool Material Types](#) for more information.
- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Deleting Tool Material Types

If a given tool material type is not used by any tool shop, you can delete the tool material type by following these steps:

- 1 In the CMWB navigation pane, expand **Global Cost Model Information** and then expand **Tool Material Types**.
- 2 Right click the desired tool material type, and select **Delete Object** from the context menu.

Working with Process Setup Options

Each process, operation, and branch node can have one or more associated setup options, which allow the end user to input data that is specific to the current part. A CSL module can access setup option data with the CSL standard input `setup`, which has one field for each setup option of the module's associated node.

The following example accesses `nominalWallThickness`, which is a setup option for the Injection Molding process in the Plastic Molding process group:

```
nominalWallThickness = { _
    setup.nominalWallThickness if setup.nominalWallThickness != null
    _
    3 otherwise _
}
```

A setup option can have multiple input modes—see the description of mode tables in [About Process Setup Options](#). The CSL standard input `setup` also has a field that allows determination of the currently active mode. The field name consists of `Mode` appended to the setup option name. Here is an example that accesses the mode for `numberOfDrops` (number of hot drops), which is another setup option for the Injection Molding process in the Plastic Molding process group:

```
costPerHotDrop = { _
    plant.costPerStandardHotDrop if setup.numberOfDropsMode ==
    'userStandard' _
    plant.costPerSVGHotDrop if setup.numberOfDropsMode == 'userSVG' _
    0 otherwise
}
numHotDrops = setup.numberOfDrops * numCavities
numHotDropsPerCavity = setup.numberOfDrops
```

See [CSL Language Overview](#) for more information on CSL.

This section covers the following topics:

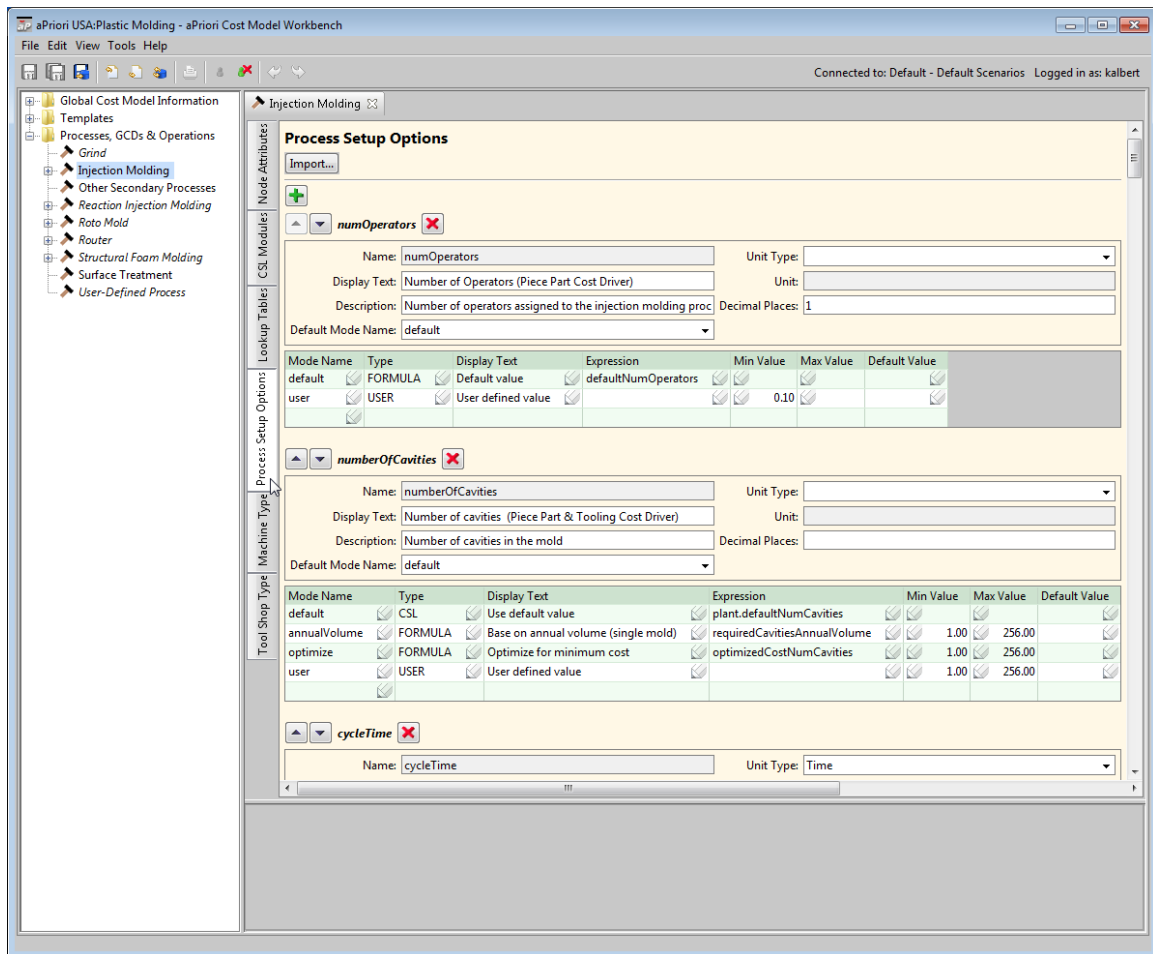
- [Navigating to Process Setup Options](#)
- [About Process Setup Options](#)
- [Modifying Process Setup Options](#)

- Adding and Deleting Process Setup Options

Navigating to Process Setup Options

Follow these steps to navigate to the process setup options for a given process or operation:

- 1 In the CMWB navigation pane, expand **Processes, GCDs & Operations**; double-click the desired process, or navigate to the desired operation and double-click it.
- 2 In the editing pane, select the **Process Setup Options** tab. The editing pane displays information about the specified node's setup options. See [About Process Setup Options](#).




To navigate to the setup options for a given branch node, follow these steps:

- 1 In the CMWB navigation pane, expand **Templates**, navigate to the desired branch node, and double-click it.
- 2 In the editing pane, select the **Process Setup Options** tab. The editing pane displays information about the specified node's setup options. See [About Process Setup Options](#).

About Process Setup Options

For each setup option, the editing pane displays the following:

- Arrows, , that allow you to control the order in which the options appear in aPiori's end user interface.
- Name of the setup option
- X symbol for module deletion
- Option definition fields:
 - **Name:** name for the field of `setup` that corresponds to this setup option. Cannot be modified.
 - **Display Text:** name for this setup option that appears in the end user interface.
 - **Description:** optional description of this setup option.
 - **Default Mode Name:** name of the mode that is active by default—see the description of mode tables, below.
 - **Unit Type:** type of unit applicable to the option's values.
 - **Unit:** unit for the option's value. This is determined by the unit type, and is not directly editable.
 - **Decimal Places:** precision to which the end user specifies this option's value.

The **Display Text** and **Description** fields can use HTML text markup formatting syntax. For example, surround text with `` and `` to format the display in bold text.

In addition, each setup option has a mode table, which contains one row for each input mode presented to the end user. If there are multiple nodes, the end user chooses an input mode by clicking a radio button. Each row of the mode table contains the following fields:

- **Mode Name:** string value of the mode field of `setup` for this mode.
- **Type:** specifies the type of input mechanism presented to the end user, and specifies how the option value is determined. When you double click in this field, the following types appear in the dropdown menu:
 - **CSL:** presents no further input mechanism (other than this mode's radio button). The option's value is the value of the CSL expression specified in the **Expression** field of the mode table. The expression must not reference CSL standard inputs.
 - **FORMULA:** presents no further input mechanism (other than this mode's radio button). The option's value is the value of the formula specified in the **Expression** field of the mode table. The formula must be CSL output formula.
 - **USER:** presents a data entry field. The value entered by the end user specifies the value of the setup option.
 - **LIST:** presents a dropdown list. The list's items are specified by the comma-separated list (see the figure below) in the **Expression** field of the mode table. The item chosen by the end user specifies the string value of the setup option.

Mode Name	Type	Display Text	Expression
user	LIST	User Override	None, Nickel, Chrome, Nitride, Teflon, Tribo-Coating

The list's items can also be specified by a select expression that queries a lookup table. See [Adding a Setup Option—Using List Mode to Access a Lookup Table](#) for an example.


- **I18N LIST**: presents a dropdown list. The list's items are specified by the comma-separated list of string IDs in the **Expression** field of the mode table. The language-specific strings corresponding to the IDs are presented to the end user. The string item chosen by the end user specifies the string value of the setup option.
- **BOOLEAN**: presents a checkbox to the end user. If checked, the option's value is `true`; the option's value is `false` otherwise. You must specify a default value for this mode. If the default is `true`, the checkbox is checked initially; if the default is `false`, the checkbox is initially unchecked.
 - **Display Text**: text displayed for this mode.
 - **Expression**: see the description of the **Type** field of the mode table, above.
 - **Min Value**: for numeric input, the minimum value that the end user can enter.
 - **Max Value**: for numeric input, the maximum value that the end user can enter.
 - **Default Value**: value used for this option if the end user does not specify a value.



The editing pane also displays a + icon and **Import** button, which allow you to add new setup options—see [Adding and Deleting Process Setup Options](#).

For examples, see [Adding New Process Setup Options](#) in [Common Task Examples](#).

Modifying Process Setup Options

Follow these steps to modify a process setup option:



- 1 Navigate to the setup options for the desired node (see [Navigating to Process Setup Options](#)).
- 2 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 3 You can now perform the following modifications (see [About Process Setup Options](#)):

Use the arrows,  , to change the order in which the options appear in aPriori's end user interface.

Modify an option definition field (except **Name** and **Unit**) by clicking in the field and entering a new value, or (for **Default Mode Name** and **Unit Type**) double clicking and selecting an item from the dropdown menu.

Modify a field of the mode table by clicking in the field and entering a new value, or (for **Type**) double clicking and selecting an item from the dropdown menu.




Add or remove mode table rows. Add a row by entering data in the last table row. Remove a row by right clicking the row and selecting **Remove**.

- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding and Deleting Process Setup Options


You can either create a new process setup option from scratch or create a new process setup option by copying an existing setup option.

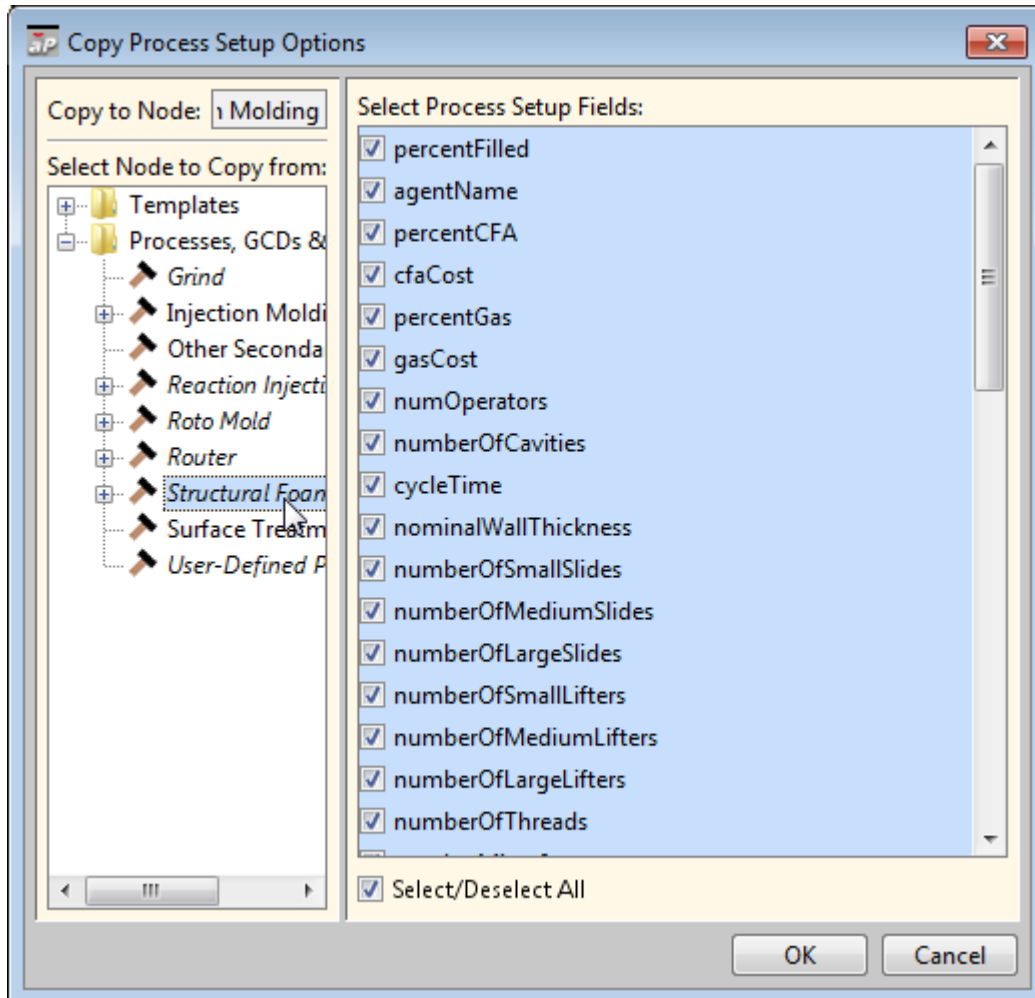
Follow these steps to create a new process setup option from scratch:



- 1 Navigate to the setup options for the desired node (see [Navigating to Process Setup Options](#)).
- 2 Select **Override Object** from the **CMWB Edit** menu, or click the override icon, , in the toolbar.
- 3 Click the green + icon. Option definition fields and an empty mode table appear at the bottom of the editing pane (see [About Process Setup Options](#)).
- 4 Enter the desired information in the mode table (see [About Process Setup Options](#)). You must add at least one mode.
- 5 Enter the desired information in the definition fields (see [About Process Setup Options](#)).
- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See also [Adding New Process Setup Options](#) in Common Task Examples.

Follow these steps to create a new process setup option by copying an existing setup option:

- 1 Navigate to the setup options for the desired node (see [Navigating to Process Setup Options](#)).
- 2 Select **Override Object** from the **CMWB Edit** menu, or click the override icon, , in the toolbar.
- 3 Click the Import button. The **Copy Process Setup Options** dialog appears.



- 4 In the dialog's navigation pane, select the process, operation, branch node to copy from.
- 5 In the **Select Process Fields** pane, check the setup options to copy, and click **OK**. Copies of the selected setup options appear at the bottom of the CMWB editing pane (see [About Process Setup Options](#)).
- 6 Modify the setup option as needed. See [Modifying Process Setup Options](#).
- 7 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 8 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Including and Excluding GCD Types

You can specify which GCD types aPriori attempts to recognize for a given process group. This can be helpful, for example, if you know that your organization's parts never

have a certain type of GCD. By excluding that type, you improve the performance of GCD extraction, and potentially prevent cases of erroneous extraction.

Follow these steps to enable or disable GCD types in the current cost model:


- 1 Select **Enable GCD Type(s)...** from the File menu.
- 2 Check a GCD type to enable it. Uncheck a GCD type to disable it.
- 3 Click **OK**.

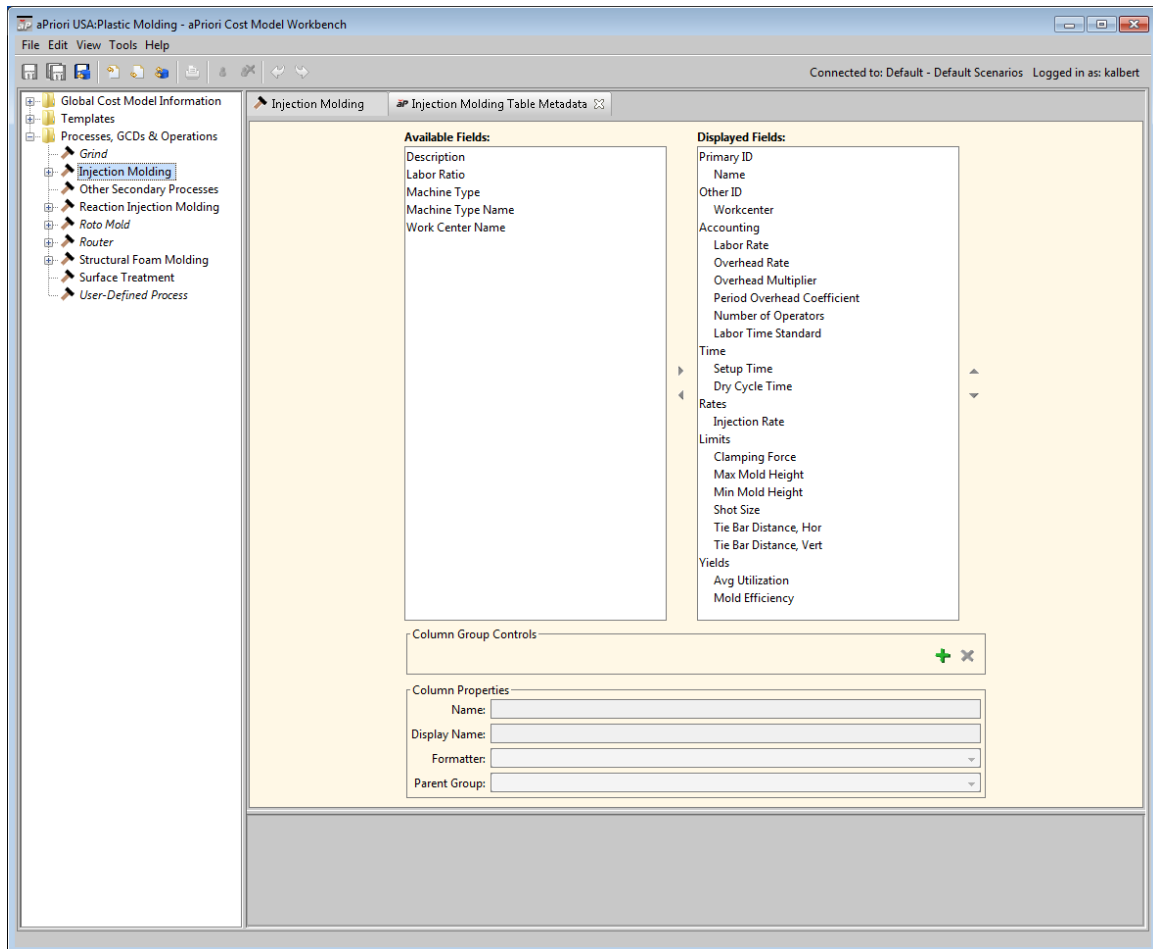
Working with Column Groups and Column Properties

Each of the following kinds of metadata tables has an associated set of column properties and column groups that you can use control the appearance of data in VPE Manager and in aPriori end user tables such as material and machine tables:

- Machine type: See [Working with Machine Metadata](#).
- Material type. See [Working with Material Metadata](#).
- Material stock type: [Working with Material Stock Metadata](#)
- Tool shop type. See [Working with Tool Shop Metadata](#).
- Tool material type. See [Working with Tool Material Metadata](#).
- Lookup table definition. See [Working with Lookup Tables](#).

Follow these steps to manage column properties and column groups:

- 1 Navigate to a type table.
- 2 Click the  icon to the left of the table name in the editing pane. The column properties and column groups page appears in a new tab in the editing pane.





The data table column names appear under **Available Fields** and **Displayed Fields**. These are the displayed names that correspond to the metadata table values of **Field Name**. Only those under **Displayed Fields** are displayed in VPE Manager and aPriori end user data tables.

To view and modify a column's properties, select the column name from **Available Fields** or **Displayed Fields**. The following fields appear under Column Properties

- **Name:** value of Field Name for this column.
- **Display Name:** displayed name of the column.
- **Formatter:** controls the format of displayed values.
- **Parent Group:** column group in which this column appears. In VPE Manager tables and aPriori end user tables, column groups can be expanded by clicking the plus sign.

To change which columns are displayed, select a column or group and use the left and right arrows, ◀ and ▶. Note that if you select a group and click the left arrow, all the columns in that group are moved out of **Display Fields**. Note also that whenever you move a column into **Display Fields**, you must re-specify its Parent Group.

To change the order in which columns appear in VPE Manager and aPriori end user tables, select a column or group, and use the up and down arrows, ▲ and ▼.

Create and delete column groups with  and .

- 3 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

Working with Node Attributes

Node attributes serve a variety of purposes. Some predefined node attributes directly affect cost engine behavior (as with, for example, `utilizationProcess`), and others (such as `contourCuttingProcess`) merely provide a label or data item that is available to CSL modules via predefined functions such as `hasAttribute` and `getAttributeValue`--see [Node Attribute Functions](#). You can configure your cost models to make customized use of either kind of predefined node attribute. You can also define new node attributes to associate a label or a data item with a process, operation, or branch node.

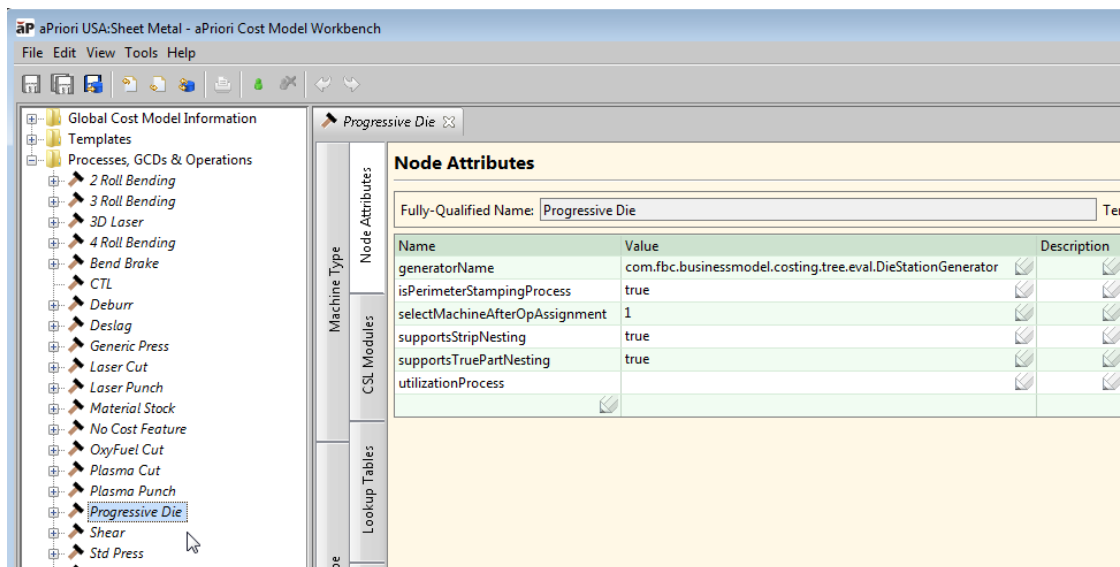
More information is provided in the following sections:

- [Navigating to Node Attributes](#)
- [Adding, Removing, and Modifying Node Attributes](#)
- [Predefined Node Attributes](#)

Navigating to Node Attributes

Follow these steps to navigate to the node attributes for a given process or operation:

- 1 In the CMWB navigation pane, expand **Processes, GCDs & Operations**; double-click the desired process, or navigate to the desired operation and double-click it.
- 2 In the editing pane, select the **Node Attributes** tab (if it isn't already selected). The editing pane displays a table of the specified node's attributes.






To navigate to the attributes for a given branch node, follow these steps:

- 1 In the CMWB navigation pane, expand **Templates**, navigate to the desired branch node, and double-click it.

- 2 In the editing pane, select the **Node Attributes** tab (if it isn't already selected). The editing pane displays a table of the specified node's attributes.

Adding, Removing, and Modifying Node Attributes

Follow these steps to modify, add, or remove a node attribute:

- 1 Select **Override Object** from the CMWB **Edit** menu, or click the override icon, , in the toolbar.
- 2 To modify an attribute, double click the **Value** field, and enter a value.
To add an attribute double click the empty **Name** field at the bottom of the attribute table, and either enter the name of the new attribute or click the down arrow and select an attribute
To remove an attribute, right click it and select **Remove**.
- 3 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Predefined Node Attributes

The node attributes listed below are used by the starting point cost models.

2SetupMilling

This attribute is used by the Machining cost model to indicate that the selected node should count as two machining operations with independent setup directions, for example, Side Milling/Side Milling or Side Milling/Facing.

aggregationOperation

The Manufacturing Process tree in aPriori Desktop displays the children of a given operation if and only if the given operation has the node attribute **aggregationOperation**.

For example, in the starting point Machining cost model, the Surface Finishing operation for a Ring GCD has a child operation for each child surface of the ring (such as Facing for a planar face that is part of the ring). Surface Finishing has the **aggregationOperation** attribute, so its child operations (such as Facing on a child planar face) are included in the Manufacturing Process tree.

The attribute only needs to be present on the operation's node—it does not need to have a value.

allowNumOccurrencesOverride

[[Define this attribute and set it to false to hide the **Number of Occurrences** context menu item.]]

assignOpsWithActivePrimaryAxesFirst

A process's child operations are considered for assignment in pass 1 if and only if the process has the value **true** for the node attribute **assignOpsWithActivePrimaryAxesFirst**.

For information on operation-assignment passes, see Operation Assignment in this Guide and Operation Assignment for Machining in the Machining chapter of the aPriori User Guide.

checkForRollingInRouting

Starting point cost models use this attribute to indicate a roll bending process. In the Bar&Tube Fab cost model, the rolling processes (3, 4, and 5 Roll Bending) each have this attribute. This is used by various processes for feasibility checks and machine selection.

For example, the process Sheet Laser Cut cannot be performed after roll bending (it can only be performed on un-rolled sheet stock). The feasibility module for Sheet Laser Cut contains the following check:

```
Rule IncompatibleTubeStock: _
  not(isNodePrecededBy(op, 'checkForRollingInRouting')) and _
  not (stock.stockForm == 'Round Tube' or stock.stockForm ==
    'ROUND_TUBE')

Message IncompatibleTubeStock: _
  'Laser Cut is not a typical process for round tube'
```

Another module that uses this attribute is machine selection for the Tube Laser process. It performs a check on either part diameter (if the stock is round or rolled) or part height (otherwise):

```
Rule MaxStockDiameterCheck: m.maxStockDiameter >= stockSize

stockSize = _
  { part.crossSection.outsideDiameter if _
    (stock.stockForm == 'ROUND_BAR' or stock.stockForm ==
    'ROUND_TUBE' or _
    stock.stockForm == 'Round Bar' or stock.stockForm == 'Round Tube'
    or _
    hasNodeInTree(op, 'checkForRollingInRouting'))
    ( max(part.crossSection.height,part.crossSection.width) )
    otherwise }
```

The attribute only needs to be present on the operation's node—it does not need to have a value.

contourCuttingProcess

The starting point Sheet Metal cost models sometimes use this attribute as an alternative to the predefined CSL function **simpleName**. The value of the attribute is the process name. Here is an example:

```
maxThicknessCutRate = {
  maxPlasmaPunchThicknessCuttingRate if _
    getAttributeValue(op, 'contourCuttingProcess') == 'PlasmaPunch'
  or _
    getAttributeValue(op, 'contourCuttingProcess') == 'LaserPunch' _
  maxWaterjetThicknessCuttingRate if _
    getAttributeValue(op, 'contourCuttingProcess') == 'WaterjetCut' _
  maxThicknessCuttingRate otherwise }
```

coring

This attribute is defined on nodes for RingedHole coremaking in the Casting cost model.

costModeClass

This node attribute is used by the Plastic Molding starting point cost model to specify that the Java class **com.fbc.businessmodel.costing.CavityCostModelM** should be used to optimize the number of mold cavities when the user selects **optimize** mode for the PSO **numberOfCavities**.

For any cost model, aPriori uses this class to calculate the number of cavities if both the following hold:

- A branch node of the current process routing defines this attribute and sets its value to **com.fbc.businessmodel.costing.CavityCostModelM**.
- Some process in the current routing defines a PSO **numberOfCavities** and the user has selected the mode **optimize**.

In this case, aPriori considers each row of the global lookup table **layoutNumCav**. It performs costing using the cavity layout specified by each table row in turn, setting the PSO (**setup.numberOfCavities**) to the number of cavities specified by the layout. aPriori treats each such costing as if it were for a distinct process-level routing, and selects the one with the lowest cost.

If you create a custom cost model with this attribute, note the following:

- The attribute must be defined on a branch node of the Component template.
- During consideration of each row of the lookup table, the PSO mode is set to **user** (this is visible from the debugger at a break point).
- aPriori sets **optimizeCostNumCavities** to the optimal number of cavities. For the PSO **numberOfCavities**, you should define **optimize** mode with type **FORMULA** and formula **optimizeCostNumCavities**, so that the Process Options Editor UI reports the optimal number of cavities.

costPerimetersFirst

If this attribute is defined on a node and set to **true**, the cost engine evaluates the assignment of Perimeter GCDs in Pass 1 before any other GCDs are costed.

Starting point cost models use this to disable Machining operations on Planar Face and Curved Wall GCDs when they lie on a Perimeter GCD that has been assigned to Waterjet Cutting. This is governed by the status of a **perRouting** cache set in **ofrPerimeterWaterjetCutting.csl**, which must be evaluated before any other Machining operations are assigned in Pass 1.

disableRingedHoleGeneration

If this attribute is defined on a node in a Machining routing, RingedHoles are not created.

displayOperationCategories

This attribute associates a cycle time subtotal class with a node. The cycle times for all nodes with a given class are aggregated together to form a displayed subtotal of total cycle time.

displayInSPDialog

A process node is available in the Cost Guide as a Secondary Process only if this attribute is defined and set to **true** for the node.

finishGrinding

This attribute is used by the Machining cost model to identify node that is either Finish Grinding, Net Shape Machining, ASSY Wall Finishing, Finish Grinding, Finishing, or Hole Grinding. It is used for finish machining rules.

generateRingedHoles

If this attribute is defined on a top-level branch node of the Component template and is set to **true**, geometry extraction creates RingedHole GCDs when appropriate. If the attribute is set to **false**, extraction does not create RingedHole GCDs.

generatorName

Some cost models use this attribute to specify a Java class that the cost engine uses to generate instances of a special operation, such as Setup (for Machining) or Die Station (for Progressive Die). See [Working with Zero-or-More Nodes](#) for more information.

hasPiercing

Not currently used.

hasPiloting

Not currently used.

hasPilotingOp

Not currently used.

hasSetupAxisKey

The Machining cost model defines this attribute on nodes that support setup axes and setup axis keys.

ignoreUpperGtolBound

This node attribute has been deprecated.

inFoundryProcess

Starting point Casting cost models use this attribute to identify processes for which scrapped parts can be re-melted and reused instead of counting towards scrap parts for the purposes of calculating Final Yield.

isBarFeedLathe

Starting point cost models use this attribute to identify bar feed lathe processes in Machining:

- 2 Axis Bar Feed Lathe with Sub Spindle
- 3 Axis Bar Feed Lathe with Sub Spindle

This attribute is used interchangeably with `requiresSingleFaceStockTrim`.

Routings using a bar feed lathe should always choose a parting operation for face rings that are children of stock trim gcds.

Here is some sample CSL that uses this attribute:

```
Rule BarFeedLatheOnly: not(isTrue(isBarFeedLatheOnly))
Message BarFeedLatheOnly: 'Roughing/Finishing is not applicable for
    face rings when the routing includes a bar feed lathe process'

isBarFeedLatheOnly = { _
    true if hasNodeInTreeWithTrueValue( op, 'isBarFeedLathe' )_
    and gcd.approachFromName == 'FACE_B'_
    and gcd.parentArtifact.artifactTypeName == 'StockTrim'
    false otherwise}
```

isGreenSand

The Casting cost model defines this attribute on a node for a process that creates the green sand mold in Sand Casting.

isPerimeterCut

The Machining cost model defines this attribute on the branch node Perimeter Cutting. The cost model uses this to determine whether a Perimeter Cutting process is in the current routing.

isPerimeterStampingProcess

The Sheet Metal cost model defines this attribute on process nodes that can perform perimeter stamping (such as Progressive Die, Transfer Die, Stage Tooling, Tandem Die, and Turret Press). It is used to determine whether a Blanking operation is needed.

isPlasmaCutting

The Sheet Metal cost model defines this attribute on the Plasma Cut node. It is used to determine whether a kerf width needs to be applied when calculating minimum stock size in Stock Machining.

isWaterjetCutting

The Sheet Metal cost models defines this attribute on the Waterjet Cut node. It is used to determine whether a kerf width needs to be applied when calculating minimum stock size in Stock Machining.

isRoughing

Starting point cost models sometimes use this attribute to identify an operation that performs rough volume removal. If this attribute is set to **'true'**, no tolerance-related compensation factor is applied to feed rate or cycle time for roughing operations, since desired tolerance is achieved by a subsequent finishing operation.

See also **operationCategory**.

Here is an example of its use:

```
isRoughingOpOnly = {true if getAttributeValue(op, 'isRoughing') ==  
  'true'_  
or getAttributeValue(op, 'operationCategory') == 'Roughing'  
false otherwise}
```

MachiningNode

This attribute is defined on the node named "Machining".

mcdType

Some special cost drivers (MCDs) are recognized during costing subsequent to geometry extraction. These are shown in the Geometric Cost Drivers pane, but only after costing. The MCDs recognized for a given routing are governed by the value of this attribute for the first node (going left to right) that defines the attribute.

Following are the supported attribute values:

- Stock Machining
- Assembly
- Turning
- Plastic Molding

- Stage Tooling

Each value is associated with one or more MCDs. Stock machining, for example, is associated with the MCD Ringed Hole, as is Turning (but see [generateRingedHoles](#)). Plastic Molding is associated with MCDs for slide and lifter bundles.

operationCategory

Starting point cost models use this attribute to identify operations in three categories:

- Roughing
- Finishing
- Holmaking

The attribute is used to aggregate the operation types for display in custom process outputs in Machining. Here is an example of its use:

```
ba_totalRoughingOperations3 = _
  select sum(op.formulaResults.cycleTime) from allSetupOps op where
  _
  op.formulaResults.cycleTime != null _
  and getAttributeValue( op, 'operationCategory' ) == 'Roughing'
```

This attribute is also sometimes used to ensure that no tolerance-related compensation factor is applied to feed rate or cycle time for roughing operations, since desired tolerance is achieved by a subsequent finishing operation. See also [isRoughing](#).

otherSecondaryProcesses

The cost model for Other Secondary Processes defines this attribute on the Packaging node. It is used to determine whether Other Secondary Processes is included in the UGC Sheet Metal routing. If it is in the routing, Part Length, Part Width, and Part Height are required user inputs.

PartiallyObstructedFaceMilling

The Machining cost model allows an obstructed setup to be used for an operation whose node defines this attribute.

perimeterCutting

Used to determine the appropriate feed rate for perimeter cutting processes (such as Plasma Cut and Waterjet Cut).

Plasma Cutting

The Sheet Metal cost model defines this attribute on the on the Plasma Cutting node of the Perimeter GCD in the Plasma Cut process. This is used by Stock Machining to determine whether the plasma cutting process is in the current routing.

PreviouslyCountersunk

Cost models define this attribute on nodes that are able to perform countersinking without any additional machining operations.

PreviouslyThreaded

Cost models define this attribute on nodes that are able to perform threading without any additional machining operations.

primaryGtolOp

Cost models define this attribute on default processes and default operations for creating finished holes or surfaces.

primaryHolemakingOp

Cost models use this attribute to help determine whether a hole finishing operation (such as reaming or boring) is required for a given GCD. This attribute is defined on each primary (that is, non-finishing) holemaking operation in Bar&Tube, Casting, Forging, Machining, Powder Metal, and Sheet Metal.

When aPirori considers whether to include a given hole finishing operation in the current operation sequence, it checks each upstream operation with this attribute, to see if one is capable of achieving the required tolerance for the current GCD. If one can, the finishing operation is not included.

When aPirori calculates cycle time for a primary holemaking operation, if a finishing operation is included in the operation sequence for the current GCD, no tolerance-related compensation is applied, since the extra time to achieve the required tolerance is incurred by the finishing operation. If there is no finishing operation, tolerance-related compensation *is* applied to the calculation of cycle time for the primary holemaking operation.

requiresRotationalAxes

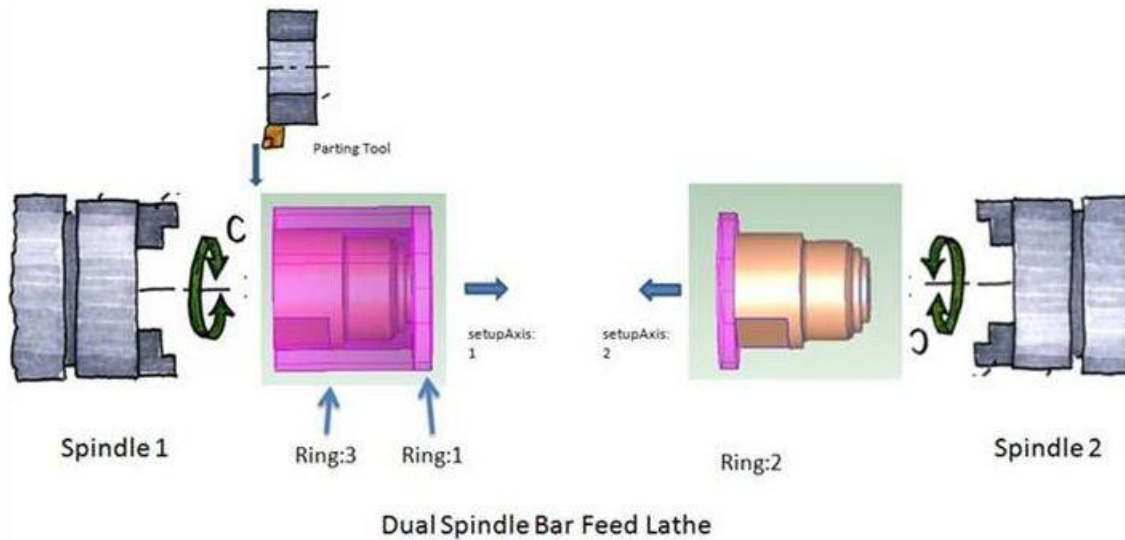
Starting point Machining cost models use this attribute to identify some processes that use rotational setup axes (4 and 5-Axis Mill).

requiresSingleFaceStockTrim

The starting point Machining cost models define this attribute on the following processes:

- 2 Axis Bar Feed Lathe with Sub Spindle
- 3 Axis Bar Feed Lathe with Sub Spindle

When this attribute is defined for a node, a custom **stockTrim** cost driver is created with only two rings: one FACE ring and one OD ring. For the part in the figure below, aPirori generates either **Ring:1** or **Ring:2** (but not both) along with **Ring:3**.



The attribute is also used in Stock Machining to determine material stock length. Face stock length allowance only needs to be accounted for on one face instead of two:

```
minStockLength = {part.minStockLength _
  + faceStockLengthAllowance if isBarFeedLatheInRouting _
  part.minStockLength + 2 * faceStockLengthAllowance otherwise}
```

```
defaultStockLengthAllowance = { partingRingWidth if
  isBarFeedLatheInRouting
  defaultStockAllowance otherwise }
```

```
isBarFeedLatheInRouting = _
  hasNodeInTreeWithTrueValue(op, 'requiresSingleFaceStockTrim')
```

runFourthCostingPass

Process or branch nodes that define this attribute and set its value to `true` are evaluated in the fourth operation-assignment pass. For information on operation-assignment passes, see Operation Assignment in this Guide and Operation Assignment for Machining in the Machining chapter of the aPriori User Guide.

selectMachineAfterOpAssignment

For a process node that defines this attribute and sets it to a non-null value, the cost engine evaluates the machine selection module during the yield pass (prior to the first evaluation of the process taxonomy module, but after operation assignment). In this case, `childOps` and `allOps` are in scope within the machine selection CSL, so that machine selection can take into account which operations (or in the case of Progressive Die, for example, how many die stations) are assigned.

SimpleHoleFinishing

The Machining cost model defines this attribute on the node called "Finish Machining".

sortGcdsByFinishArea

This attribute affects operation assignment for a process node or branch node that defines this attribute and sets its value to `true`. For such a node, operation assignment in passes 3 and 4 considers surfaces in order of descending size of the property `areaFinished`.

For information on operation-assignment passes, see Operation Assignment in this Guide and Operation Assignment for Machining in the Machining chapter of the aPriori User Guide.

stockPrep

Stock Machining and Machining starting point cost models use the presence or absence of this attribute to determine if the current routing should include a separate or an integrated Stock Prep process.

Here is an example of its use:

```
Rule CreateSeparateStockPrepProcess: _
not (isTrue (plant.createSeparateStockPrepProcess)) and _
not (hasStockPrepNodeIncluded)
Message CreateSeparateStockPrepProcess: _
'plant.createSeparateStockPrepProcess set to create separate stock
  prep processes or stock prep process manually selected'

stockPrepNode = getNodeInTree (op, 'stockPrep')
hasStockPrepNodeIncluded = { true if ( stockPrepNode!=null and _
stockPrepNode.inclusionStatus==InclusionStatus.USER_INCLUDE )
false otherwise }
```

supportsStripNesting

Starting point Sheet Metal cost models define this attribute and set it to `true` for processes that support strip nesting. This attribute affects whether strip nesting occurs only if the current utilization mode is True-Part Shape Nesting. Other utilization modes are unaffected by the value of this attribute.

supportsTruePartNesting

Starting point Sheet Metal cost models define this attribute and set it to `true` for processes that support true part-shape nesting. Setting this attribute to `false` for such a process has the following effects on routings that include that process:

- The material utilization mode defaults to rectangular nesting.
- The true part-shape nesting option is omitted from the Material Utilization section of the Material Selection dialog.

threading

Cost models define this attribute on the node called "threading".

toolMaterialKey

In the Casting starting point cost model, the following processes set this attribute to either `moldingMaterial` or `patternMaterial`:

- Die Casting
- Horizontal Automatic
- Manual Floor Moldmaking
- Manual Std Moldmaking
- Vertical Automatic

useAdvancedUtilization

Setting this attribute to `true` causes the cost engine to use special internal procedures to evaluate the formulas `utilization` and `numPartsPerSheet`. Any CSL for these formulas is ignored.

Sheet Metal starting point cost models use this attribute for processes that support special non-CSL part-nesting analysis (which includes true part-shape nesting as well as other nesting modes). If the attribute is not defined, or is set to false, the CSL is used to evaluate these formulas.

In some nesting modes, the UI shows a nesting diagram for processes that set this attribute to `true`.

userNamedProcess

Every process group includes a process, User-Defined Process, which defines this attribute. When a new process group is created, such a process is added automatically.

useSheetStock

The Bar&Tube Fab starting point cost models set this attribute to `true` to designate a utilization node (see `utilizationProcess` below) whose stock should be selected from sheet stock (rather than bar & tube stock). For the utilization node of roll bending routings, the cost model defines this attribute and sets it to `true`. This is used internally by aPriori to guide stock selection. All stock sizes are determined to be virtual stock. Custom cut mode is not valid for sheet stock.

utilizationProcess

This attribute is used to designate a stock selector and materials utilization nodes. In a given routing, aPriori evaluates the material stock selection module for (and only for) the designated stock selector node. In addition, aPriori evaluates utilization for (and only for) the designated utilization node of a given routing.

The simplest way for a routing to designate the stock selector node is by the presence of the node attribute `utilizationProcess`, where the attribute has no value. If there is more than one node with this attribute (and they are all valueless), the cost engine treats the right-most active one as the stock selector node.

A routing can also designate the stock selector node by specifying the node's name as the value of **utilizationProcess** for some other node in the routing. If there is more than one node with a value for this attribute, the left-most one specifies the name of the stock selector node.

For example, for the sheet metal process group, if a routing includes the branch node `Prog Die`, which has a node attribute **utilizationProcess** whose value is `Progressive Die`, the cost engine evaluates the stock selection module associated with `Progressive Die` (assuming no other node has a value for **utilizationProcess**).

If the stock selection module establishes a collection of stocks as the value for the CSL output `validMaterialStocks`, the cost engine considers each stock in the collection.

If the stock selection module does not establish a non-null value for the CSL output `validMaterialStocks`, the cost engine considers only one, virtual stock, specified as follows:

- For the Machining process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:
 - Length:
 - CSL output `virtualStockLength`, if non-null.
 - Otherwise, the larger of the current part's `minStockLength` and the value of the plant variable `standardStockLength`, if the plant variable is defined.
 - Otherwise, the larger of the current part's `minStockLength` and 20 feet.
 - Width:
 - CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the width of the current part's cross-section.
 - Height:
 - CSL output `virtualStockHeight`, if non-null.
 - Otherwise, the height of the current part's cross-section.
 - Inside diameter:
 - CSL output `virtualStockInsideDiameter`, if non-null.
 - Otherwise, the inside diameter of the current part's cross-section.
 - Outside diameter:
 - CSL output `virtualStockOutsideDiameter`, if non-null.
 - Otherwise, the outside diameter of the current part's cross-section.
 - Thickness:
 - CSL output `virtualStockThickness`, if non-null.
 - Otherwise, the thickness of the current part's cross-section.
- For the Sheet Metal process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:
 - Length:
 - The larger of the blank's length and the CSL output `virtualStockLength`, if non-null.

- Otherwise, the larger of the blank's length and the value of the plant variable `standardStockLength`, if the plant variable is defined.
- Otherwise, the larger of the blank's length and 8 feet.
- o Width:
 - The larger of the blank's width and the CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the larger of the blank's width and the value of the plant variable `standardStockWidth`, if the plant variable is defined.
 - Otherwise, the larger of the blank's width and 4 feet.

Waterjet Cutting

The Sheet Metal cost model defines this attribute on the Waterjet Cutting node of the Perimeter GCD in the Waterjet Cut process. This is used by Stock Machining to determine whether the waterjet cutting process is in the current routing.

3 Working with Cost Model Logic

All cost models have the same high-level logic, which defines both the flow and effects of module evaluation (see Cost Engine Details). The detailed logic of a cost model is specified by routing templates together with the CSL modules associated with the templates' nodes, as well as by global CSL modules. The code in a CSL module specifies the rules and formulas that make up a particular portion of a cost model, such as the portion that computes a cost taxonomy for a particular process. This chapter describes how experienced users can employ CSL and routing templates in order to implement and customize the details of cost model logic.

This chapter covers the following topics:

- CSL Language Overview
 - Viewing and Editing CSL Modules
 - Creating and Deleting Processes, Operations
 - Template Pruning
 - Material Stock Selection
 - Process and Operation Optionality
 - Process and Operation Feasibility
 - Machine Selection
 - Tool Selection
 - Process and Operation Taxonomy
 - Working with Templates
-

CSL Language Overview

When the cost engine evaluates a CSL module, it either calculates values for the module's *outputs*, such as `cycleTime` and `laborCost`, or it returns a boolean value, `true` or `false`. Output values become available to other CSL modules during the costing process, as well as to the aPriori GUI when costing is complete. Returned boolean values help guide the choice of process routings and operation sequences.

CSL modules can be divided into three categories:

- *Taxonomy modules* calculate those outputs, such as `cycleTime` and `laborCost`, that are specified by the module's associated formula table in the CMWB (see [Navigating from the Template Graph to the Data for a Given Node](#)). Taxonomy modules include the following module types:
 - Process taxonomy
 - Operation taxonomy
- *Selection modules* establish values for special output identifiers associated with their module types, for example, `machine` for machine selection modules and `tool` for tool selection modules. Other modules can subsequently use these identifiers to access the established values. Selection modules include the following module types:
 - Material stock selection
 - Machine selection
 - Tool selection
- *Modules that return a boolean value*, such as feasibility modules, consist primarily of rules. They return `true` if all the rules succeed; they return `false` if any rule fails. Modules that return a boolean value include, among others, the following module types:
 - Process feasibility
 - Operation feasibility
 - Template pruning
 - Process optionality
 - Operation optionality

In addition to outputs, CSL modules also have a number of *standard inputs*, such as `part`, `material`, `machine`, `plant`, and `setup`, which allow a module's code to access VPE and cost model data, such as plant variables, current setup options, and attributes of the current part, machine, and material, among other values. See [Inputs](#) in Cost Scripting Language Reference for more information on standard inputs.

This language overview includes the following sections:

- [Formulas](#)
- [Rules](#)
- [Imports](#)
- [Values](#)
- [Expressions](#)
- [Line Continuation](#)

- Comments
- Using the CSL Debugger
- CSL Reference Information

Formulas

The CSL language is centered around formulas and rules. Formulas establish values for a module's outputs.

A formula is essentially a named expression. Here is a typical example from aPiori's plastic molding starting points:

```
computedCycleTime = (injectTime + coolTime + ejectTime ) /
  numCavities
```

In this formula, `injectTime`, `coolTime`, `ejectTime`, and `numCavities` (number of mold cavities) refer to other formulas in the same module. In the course of evaluating a given formula, the cost engine evaluates the formulas and rules referred to by the given formula. Here is the formula for `coolTime`:

```
coolTime = 0 - _
  (((nominalWallThickness)^2)/(2*constants.pi*thermalDiffusivity) _
  *ln((constants.pi/4)*(ejectDeflectionTemp - moldTemp)/ _
  (meltingTemp - moldTemp)))
```

Here, `nominalWallThickness` is derived from a setup option. `constants` is a standard input that provides access to useful constants such as `pi`. `thermalDiffusivity`, `ejectDeflectionTemp`, `moldTemp`, and `meltingTemp` are all derived from attributes of the current material, with the following formulas:

```
thermalDiffusivity = material.thermalDiffusivity
moldTemp = material.moldTemp
meltingTemp = material.meltingTemp
ejectDeflectionTemp = material.ejectDeflectionTemp
```

In these formulas, `material` is a standard input. It has a field (such as `meltingTemp`) for each material attribute (as listed in the **Material Composition** table of aPiori's **Material Selection** dialog). CSL code accesses fields of an input with the dot notion shown above.

Note that the cost engine generally evaluates only those formulas that must be evaluated in order to assign values to the module's outputs (but see [Set Blocks](#) in Cost Scripting Language Reference). So some formulas in a module might never be evaluated.

In addition to formulas, CSL also supports function definitions, which are essentially parameterized formulas. Here is an example:

```
GetLaborCost(laborTime, laborRate) = laborRate * laborTime /
    SEC_PER_HR
```

Function invocation expressions specify actual parameters for the definition's formal parameters. CSL includes a number of predefined functions, such as `ln`, the natural logarithm function, used above in the `coolTime` formula. See [Predefined Functions in Cost Scripting Language Reference](#) for a list of predefined functions.

A formula or function definition must end with a line break. Line breaks in the middle of a formula or function definition are generally allowed only through the use of a line continuation, a space followed by an underscore, as shown in the `coolTime` formula, above.

Rules

Rules are essentially named or unnamed boolean expressions. When a rule is evaluated, the expression is evaluated, and the rule returns the result. For a named rule, the cost engine assigns the result to the name, and, if the result is false, the *message* with a matching name is displayed in the aPriori message pane. If a rule fails, and there is no message with an exactly matching name, a blank message is issued.

Here is an example from the injection molding feasibility module for aPriori's plastic molding starting point cost model:

```
Rule CompatibleMaterial: material.canIM_SFM
Message CompatibleMaterial: _
    'Failed because you cannot Injection Mold this type of material'
```

This rule uses the standard input `material`, which has one field for each attribute of the current material (as listed in the **Material Composition** table of aPriori's **Material Selection** dialog). The attribute `canIM_SFM` is set to `true` for thermoplastics and `false` for thermosetting materials. When the rule returns false, the cost engine issues the failure message.

As with formulas, rules can refer to other rules and formulas, and the cost engine evaluates the referenced rules or formulas in the course of evaluating the rule that refers to them.

For modules, such as feasibility modules, that return true or false, the cost engine evaluates the module's rules in order (except for rules that are evaluated sooner in order to evaluate another rule or formula that refers to them). If a rule evaluates to false, module evaluation terminates and the module returns false. If all the rules in a module return true, the module returns true.

Each rule and each message must end with a line break.

Advice Rules

An advice rule, like a rule, specifies a boolean expression. Each advice rule has the following form:

```
AdviceRule <rule-name> : <boolean-expression>
```

While a rule specifies a feasibility *requirement*, an advice rule specifies an *inadvisable*—but not fatal—circumstance pertaining to the current process or operation. If the expression specified by a rule evaluates to *false*, the current process or operation is deemed infeasible, a message is issued to the message tree, and evaluation of the current module ceases. In contrast, if the expression specified by an advice rule evaluates to *true*, advice is issued to the Design-to-Cost (DTC) interface, and evaluation of the current module continues. (See the *aPriori Professional User Guide* for information about Design to Cost.)

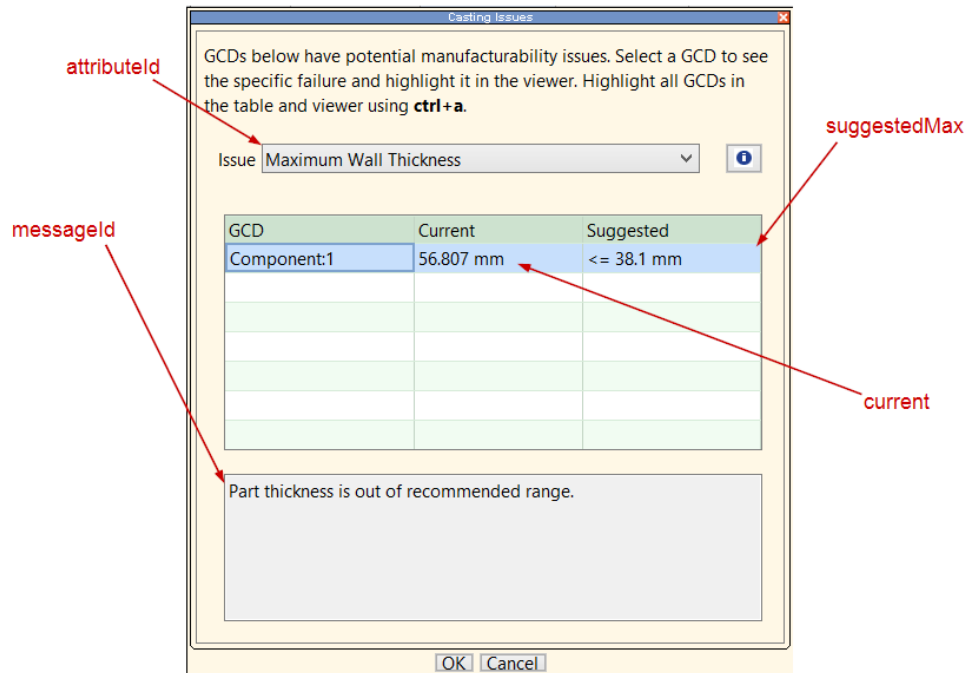
Each advice rule has an associated Advice construct, which specifies the content of the advice to be issued to the DTC interface. Here is an example of an advice rule together with its associated Advice construct:

```
AdviceRule maxWallThicknessAdviceCheck:_
    safeEval(maxPartThicknessAllowedRelevant, null) != null and _
    (partMaxThickness > maxPartThicknessAllowedRelevant)
Advice maxWallThicknessAdviceCheck: dtcMessage(
    'attributeId', 'Maximum Wall Thickness',
    'messageId', 'Part thickness is out of recommended range.',
    'current', partMaxThickness,
    'suggestedMax', maxPartThicknessAllowedRelevant,
    'unitType', 'Length',
    'custom.panelOutput', 'castingIssue')
```

This advice rule is a simplified version of an advice rule in the die casting cost model. It issues advice when the part's maximum thickness exceeds the maximum recommended thickness for the current material. The content of the issued advice is specified by a call to the predefined function **dtcMessage**. The arguments to this function consist of field-name/field-value pairs, where each field is set to one component of the advice:

- **attributeId**: the first pair of arguments sets the field **attributeId** to the string **'Maximum Wall Thickness'**. This specifies the menu item to select in order to view the category of advice that includes the current advice.
- **messageId**: the next pair of arguments sets **messageId** to the string **'Part thickness is out of recommended range'**. This specifies the message that appears below the table when the relevant table row is selected.
- **current**: the next argument pair sets the field **current** to the part's thickness, the value that violates the advice rule's associated recommendation. This value appears in the table column labeled **Current**.
- **suggestedMax**: the next argument pair sets **suggestedMax** to the maximum recommended thickness, the value exceeded by the part's actual thickness. This value appears in the table column labeled **Suggested**.
- **unitType**: the next argument pair sets **unitType** to the string **'Length'**. This is the type of units to use when displaying the values of **current** and **suggestedMax**. The display uses the default units for the specified type (millimeters in this case).

- **custom.panelOutput:** the last pair of arguments sets the field **custom.panelOutput** to the string 'castingIssue'. This specifies that the destination of the advice is the Casting Issues dialog of the DTC interface.



See the following sections for detailed information on advice rules:

- Rules
- Advice Rules
- Advice
- dtcMessage in Miscellaneous Functions

Note: in this release, advice rules can issue advice to the DTC interface for Casting—Die, Casting—Sand, Plastic Molding, and Sheet Metal. Support for other process groups will be added in future releases.

Imports

Import directives effectively include the text of a specified file in the current CSL file. They must appear on the very first lines of a module, before rules, formulas, comments, or blank lines. Imported files are typically library files. See [Navigating to Global CSL Modules](#). Here is an example:

```
import libCommonAccounting.csl
```

Import directives must end with a line break.

The global CSL file `constants.csl` is automatically imported into all CSL modules.

Values

CSL supports the following types of values

- **Arithmetic:** can be designated by expressions (see below), numeric literals such as `50` or `21.75`, or identifiers such as `roughMass` or `machine.cycleTime`.
- **String:** can be designated by expressions (see below), string literals, such as `'Material Cost'`, or identifiers such as `op.name`.
- **Boolean:** can be designated by expressions (see below), the boolean literals `true` and `false`, or identifiers such as `op.isManualOverride`.
- **Object:** can be designated by expressions (see below) or identifiers such as `part`, `gcd`, or `machine`. Each object has one or more named fields which you can access using dot notation, as in `part.volume` or `part.material`. The value of a field can be any type of CSL-supported value, including a collection or other object.
- **Collection:** can be designated by expressions (see below) or identifiers such as `childOps` or `tubLaserCutting`. Note that a cost model's lookup tables are collections; each collection element is a table row, and each collection-element field corresponds to a table column. You can retrieve a collection's elements by using `query` or `foreach` expressions.

Expressions

CSL supports the following kinds of expressions:

- Arithmetic
- String
- Boolean
- Conditional
- Function Invocation
- Query

Advanced users can also use `foreach` expressions and associative collection access—see [Foreach Expressions](#) in *Cost Scripting Language Reference*.

Arithmetic

Arithmetic expressions can be formed using binary arithmetic operators, such as `*` and `/`. Here are some examples:

```
(injectTime + coolTime + ejectTime ) / numCavities
```

```
(material.density * material.costPerKG) / 10^9
```

```
runnerVolume / ((numCavities * part.volume) + runnerVolume)
```

String

In the message clause of a rule, string expressions can be formed using the string concatenation operator `+`. Here is an example:

```
m.name + ' is not feasible. The mold base horizontal dimension ' +
_ '(' + _
roundMoldBaseX + _
'mm ) is greater than the horizontal distance between the tie
bars (' + _
maxTieBarDistanceH + 'mm)'
```

See [Example in Machine Selection](#) for this example in context.

Boolean

Boolean expressions can be formed using unary and binary logical operators such as `and` and `not`, as well as binary arithmetic comparison operators such as `==` and `<=`. Here is an example:

```
(gcd.edgeTypeName == 'ROUND' and not(isFlangedHoleEdge)) or _
(gcd.edgeTypeName == 'CHAMFER' and
not(previousCounterSunkEdge))
```

Here `gcd` is a standard input, and `isFlangedHole` and `previousCounterSunkEdge` refer to boolean-valued formulas.

See [Example in Process and Operation Optionality](#) for this example in context.

Conditional

A conditional expression evaluates to one of several alternative values, depending on the value of the boolean expressions associated with the alternative values. Here are some examples of formulas that use a conditional expression:

```
waste = { percentRun - regrindInput if percentRun > regrindInput
0 otherwise }
```

```
partLength = {setup.partX if setup.partX != null part.length
otherwise}
```

```
isFlangedHoleEdge = _
{true if _
gcd.parentArtifact != null and _
gcd.parentArtifact.artifactTypeName == 'SimpleHole' and _
gcd.parentArtifact.isFlanged == true
false otherwise _}
```

```
}

```

(See [Example in Process and Operation Optionality](#) for this last example in context.)

The entire right-hand-side of each of these formulas is a conditional expression. The conditional expression in the first formula designates the value of `percentRun - regrindInput` if `percentRun` is greater than `regrindInput`; otherwise it designates 0.

CSL supports an alternative, equivalent form for conditional expressions. Here is an example:

```
costingMessage = {
    if (safeEval(costingMessage.severity.errorOrWarning, false) ==
        false) {
        null
    } elseif (vars.results.currentCostSummary.preferredErrorMessage
        == null) {
        null
    } else {
        msg('<html><b style="color:red">', longDescriptionString, '</b>')
    }
}
```

A conditional expression can have multiple `if` or `else if` lines. It designates the value associated with the first `boolean-expression` that evaluates to true. If no `boolean-expression` evaluates to true, the conditional expression designates the value associated with the `otherwise` or `else`.

When a condition evaluates to true, the conditions that follow it are not evaluated. So the order of the conditions can affect performance. The first conditions should be those that are most likely to evaluate to true or that are inexpensive to evaluate.

Function Invocation

A function invocation expression evaluates to the result of substituting the invocation's actual parameters for the formal parameters in the corresponding function definition. Here are some examples of formulas that invoke functions:

```
laborCost = GetLaborCost(laborTime, laborRate)

laborTime = _
    GetLaborTime_PlasticMolding_InjectionMolding( _
        processTime, _
        cycleTime, _
        numOperators, _
        laborTimeStandard _
```

)

The entire right hand side of both these formulas is a function invocation.

Query

A query expression retrieves or aggregates collection elements or values. These expressions have essentially the same semantics as SQL queries. The most basic kind of query expression designates a collection that has those elements of the queried collection that meet a specified condition. Such a query expression has the following constituents:

- Expression that designates the collection to be queried
- Variable that is, in effect, bound to each collection element in turn
- Boolean expression that specifies the query's selection criterion. This expression includes the query variable. The query result includes the elements of the queried collection that meet the selection criterion.

Here is an example:

```
select * from part.childArtifacts x where isTurningAxis(x)
```

This query expression designates a collection that includes those child GCDs of the current part that are turning axes. It has the following constituents:

- `part.childArtifacts`: designates the collection to be queried.
- `x`: variable that is, in effect, bound to each element of the collection in turn.
- `isTurningAxis(x)`: selection condition that determines which collection elements form the query's intermediate result. (`isTurningAxis`, here, is a function that returns `true` if and only if the argument, `x`, is a turning axis.)

Here, `*` indicates that the query should return a collection consisting of each selected element in its entirety. Queries can also return a collection of just the values of a specified field of the selected elements. Here is an example:

```
select x.name from part.childArtifacts x where isTurningAxis(x)
```

This query expression designates a collection containing the names of the current part's turning axes. For this query, the collection of the current part's turning axes is the *intermediate* result; the final result is the collection of the names of each element of the intermediate result.

Query expressions can also include functions that return a specified element of the intermediate results (such as the first or last element) or perform some form of aggregation of the intermediate results (such as the count, or number of elements of the intermediate result). here is an example:

```
select count(x) from part.childArtifacts x where isTurningAxis(x)
```


(See [Example in Template Pruning](#) for this example in context.)

This query expression designates the number of child GCDs of the current part that are turning axes. CSL supports the following functions on intermediate results:

- `first`: returns the first element of the intermediate query results.
- `last`: returns the last element of the intermediate query results.
- `sum`: returns the sum of the values in the intermediate query results.
- `min`: returns the smallest value in the intermediate query results.
- `max`: returns the largest value in the intermediate query results.
- `count`: returns the number of elements in the intermediate query results.
- `distinct`: returns only distinct elements, eliminating duplicates; so, for example, if the intermediate results include 3, 4, 4, 5, 5, 5, 6, 7, the final result will include only 3, 4, 5, 6, 7.

Here is an example that uses the `sum` function:

```
sumOpsCycleTime = select sum(operation.cycleTime) from childOps
                    operation
```

(See [Example in Process and Operation Taxonomy](#) for this example in context.)

Here, `childOps` is a CSL standard input whose value is the collection the current process or operation's children in the process-operation hierarchy. Each element of the collection has a field for each output (such as `cycleTime`) of the element's associated CSL modules. The query designates the sum of the cycle times calculated by each child operation's taxonomy module.

Finally, queries can specify a field of the collection elements by which to order the results or intermediate results. The following query specifies that the intermediate results be ordered by the value of `workCenterOverheadRate`, from lowest to highest:

```
select first(m) from machines m _
    where _
        ClampForceCheck and _
        tieBarHCheck and _
        tieBarVCheck and _
        moldHeightCheck and _
        shotSizeCheck _
    order by m.workCenterOverheadRate
```

(See [Example in Machine Selection](#) for this example in context.)

The query designates the machine with the lowest overhead rate (that satisfies the selection criterion). The selection criterion refers to various rules. Note that the query

variable (m, in this case) can appear in rules and formulas referenced by the selection criterion.

For more examples, see [Modifying Machine Selection and Adding a Setup Option—Using List Mode to Access a Lookup Table](#) in [Common Task Examples](#).

Line Continuation

Formulas, rules, messages, function definitions, and import directives must end with a line break. Line breaks within these constructs are allowed only through the use of a line continuation (with one exception—see below). CSL supports Visual-Basic-style line continuation, a space followed by an underscore, as in the following:

```
// A space followed by the underscore is line continuation
a_bool = (part.thickness > machine.minThickness) && _
        (part.thickness < machine.maxThickness)
```

Conditional expressions can contain a line break after a boolean expression that follows `if`.

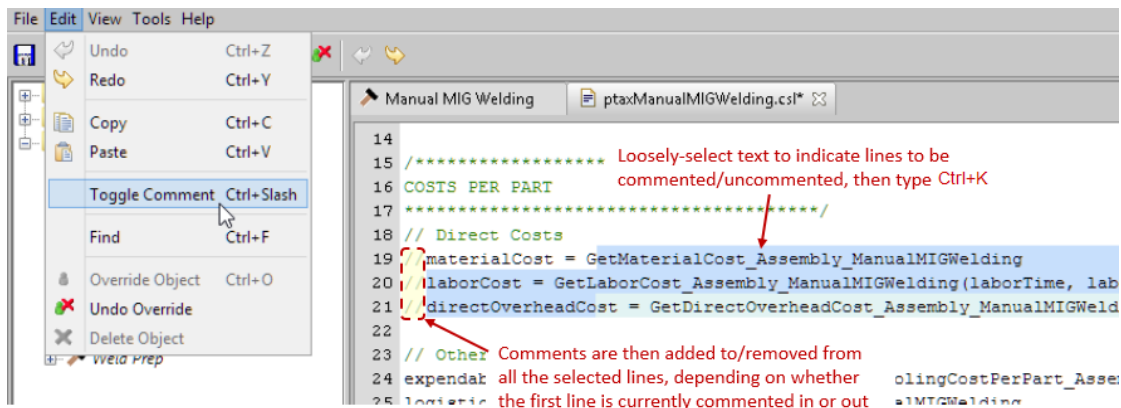
Comments

CSL supports three different forms for comments. A comment that spans lines can be enclosed between `/*` and `*/`, while a single line comment can follow either a double-slash `//` or a pound sign `#`.

You can add or remove the `//` prefix for multiple lines at once, by doing the following:

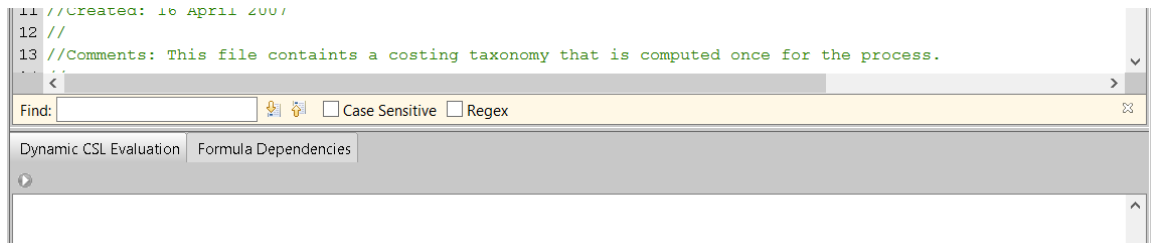
- 1 Indicate the lines to comment or uncomment by selecting at least some text from each line.
- 2 Select **Toggle Comment** from the Edit menu, or type `Ctrl+K`. Alternatively, in some locales, you can type `Ctrl+/` (that is, hold down the `Ctrl` key and press the forward slash).



If the first selected line is commented, this uncomments all selected lines. If the first selected line is not commented, this comments all selected lines.



Searching Within and Across CSL Modules

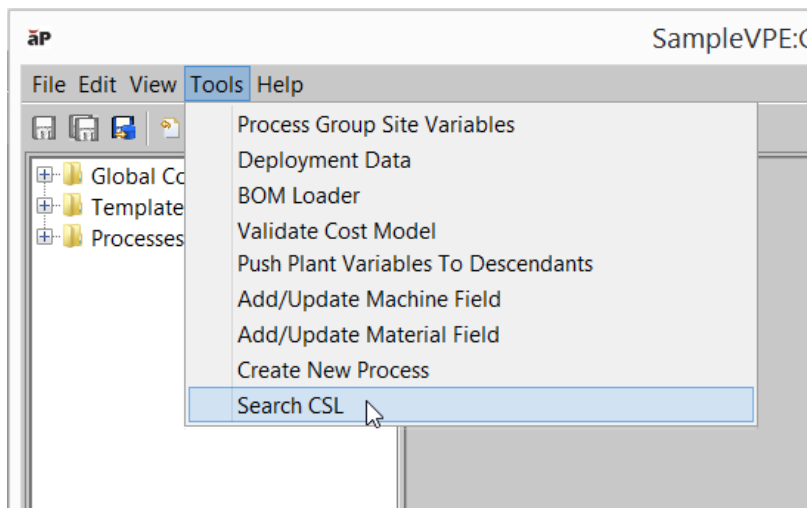
You can search within an open module by typing control-F to open a search pane below the editing pane.



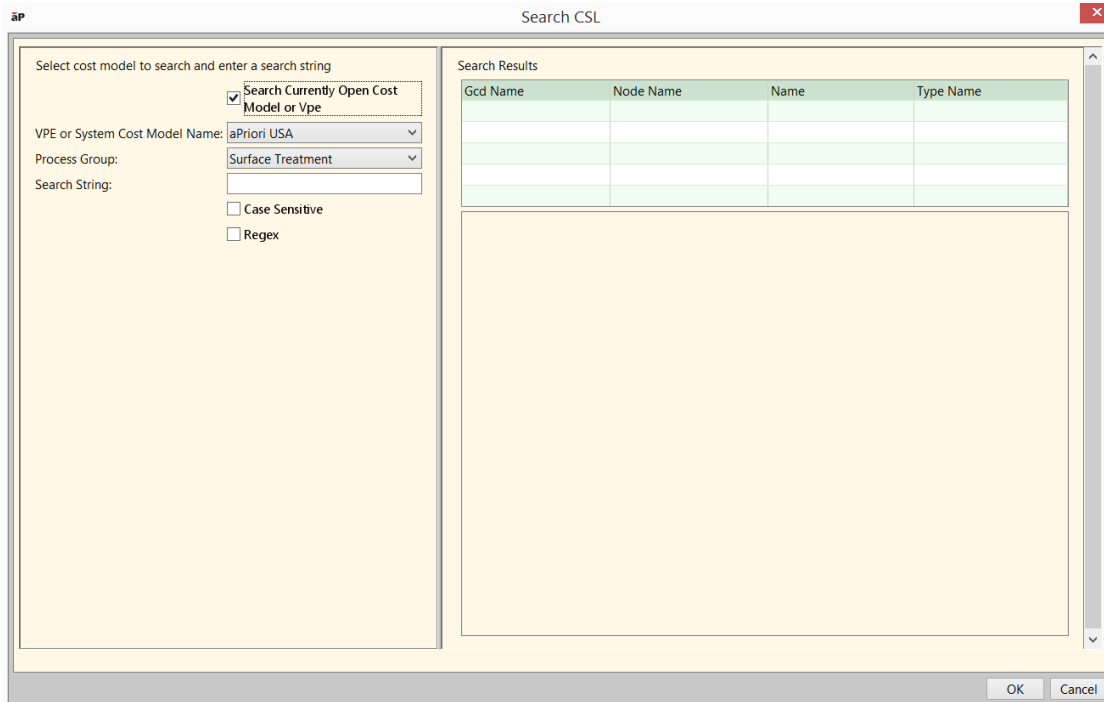
The search pane supports both case-sensitive and case-insensitive search, as well as regular expression search. You can view the search results one at a time with the next and previous buttons,  and ; alternatively use F3 for next and Shift+F3 for previous. Note that the search pane searches only within the currently-opened CSL module.

You can also use the Search CSL tool to search across all the CSL modules in a given VPE and process group. The search results list the modules in which the target string appears. Follow these steps (you can also perform these steps from the VPE Manager):

- 1 Select **Search CSL** from the Tools menu.

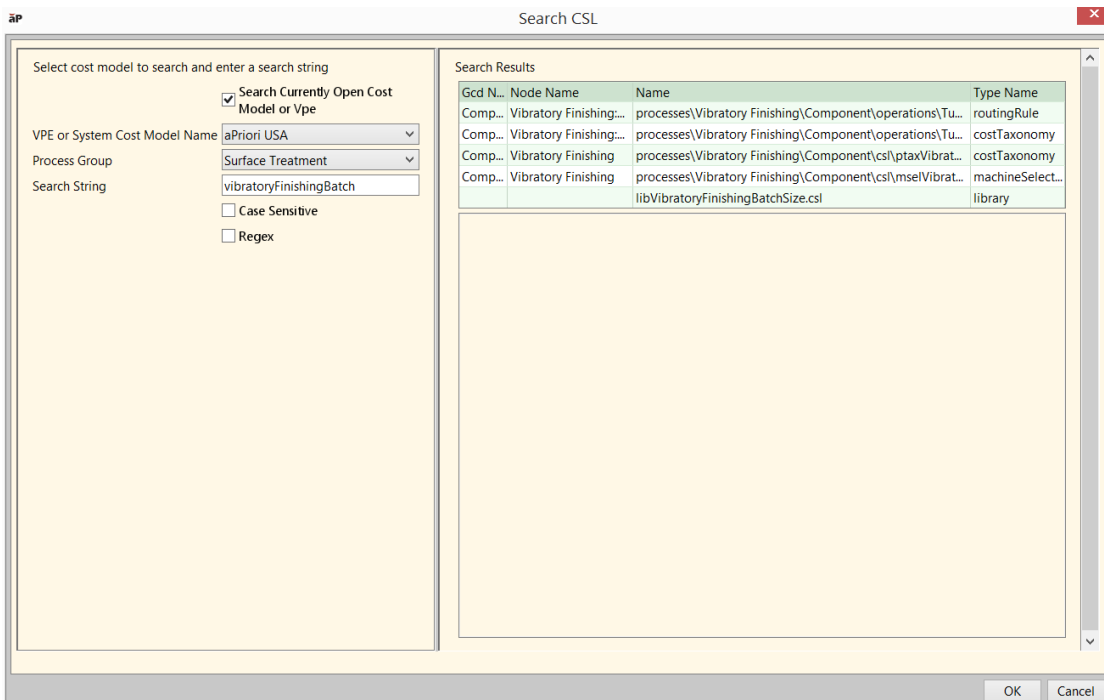


A **Search CSL** window appears:



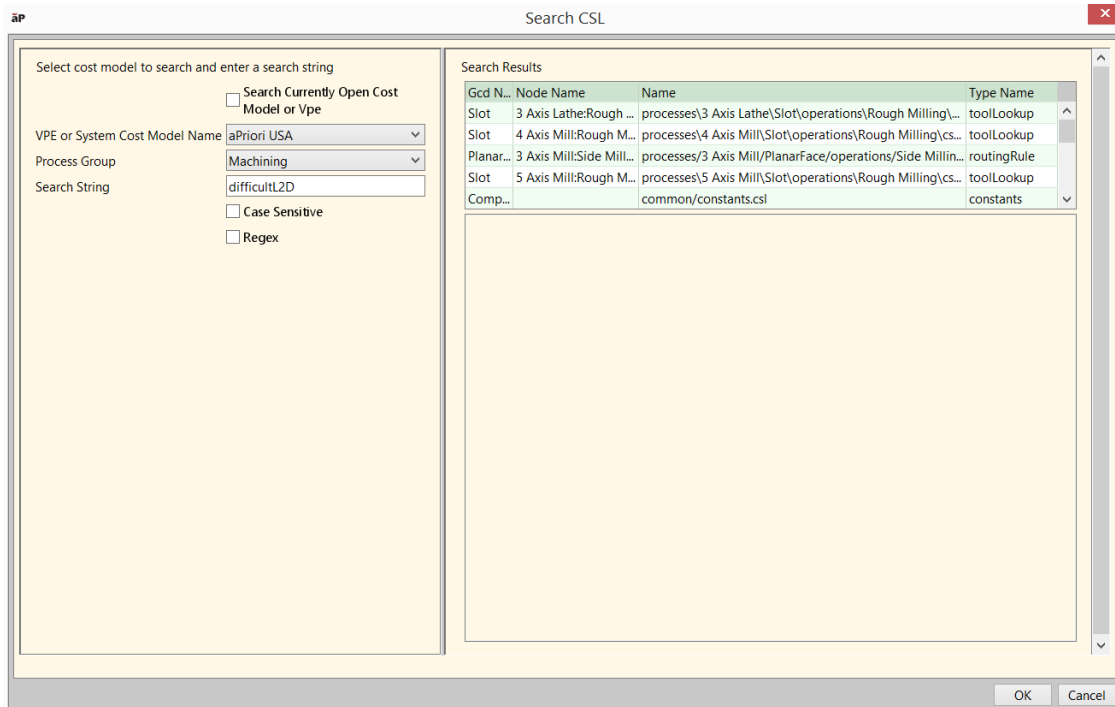
Note that you can open multiple **Search CSL** windows at once. When you select **Search CSL** from the Tools menu, a new **Search CSL** window is opened even if one is open already.

- To search the currently-open cost model, leave the top checkbox checked, and fill in the **Search String** field with a (case-sensitive) regular expression specifying the target of the search. Results are displayed and refined as you type.



To explicitly specify the VPE and cost model to search, uncheck the top checkbox, and fill in the following dialog fields:

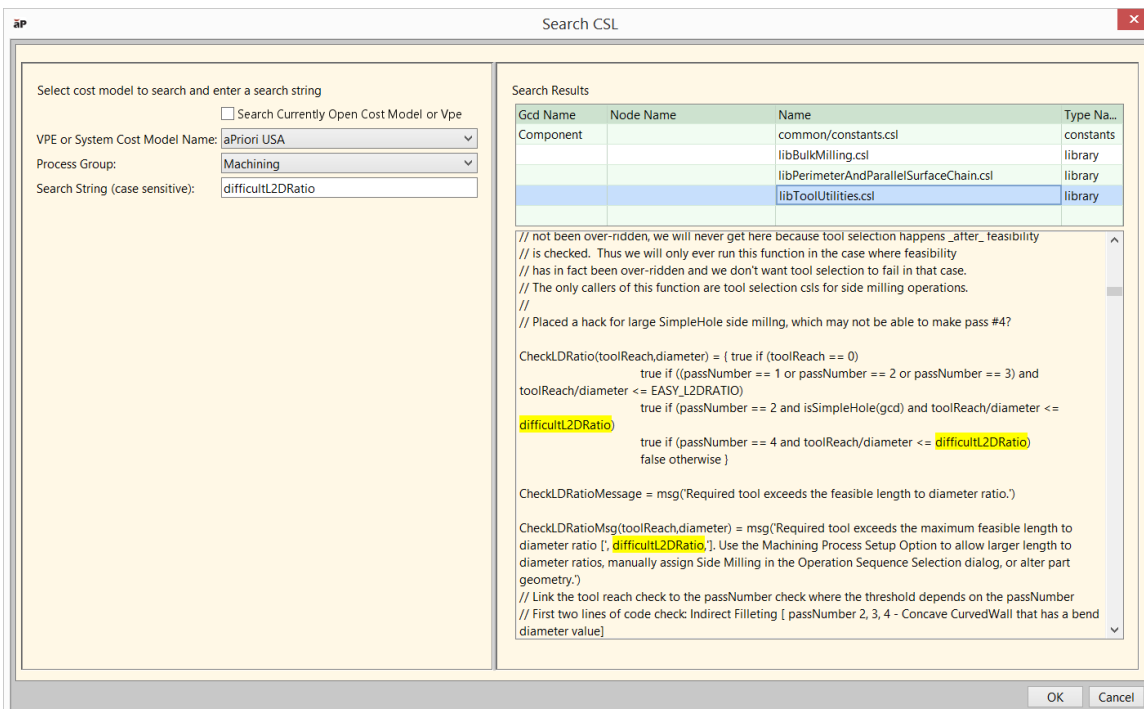
- **VPE name:** VPE to search. Setting this field populates the dropdown list of choices for the **Process Group** field.
- **Process Group:** process group to search.
- **Search String:** regular expression specifying the target of the search. Results are displayed and refined as you type.



By default, search is case-insensitive. Use the **Case Sensitive** checkbox to enable case sensitive search.

By default, search is literal. That is, by default, a search string character is never interpreted as a meta-character, such as a wildcard character. Use the **Regex** checkbox to enable regular-expression search. Note that you must use a backslash to escape regular expression meta-characters that you want to exactly match. For example, to search for strings that exactly match 'plant.', enter the search string, 'plant\.'. See <http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html> for information about regular expressions.

- 3 Click on a module in the search results to display the module's contents, with all occurrences of the target string highlighted in yellow.



Alternatively, open the result modules in the CMWB editing pane, and search within them as described at the beginning of this section.

Note that you *cannot* edit the CSL in the search results pane, and you *cannot* double-click a module name in the search results to open the module in the CMWB editing pane.


Using the CSL Debugger

The CMWB includes a debugger that allows you to suspend the flow of execution of CSL modules at a specified breakpoint, that is, at the evaluation of a specified formula or rule. When the flow of evaluation is suspended, the debugger allows you to examine the following:

- Context of evaluation of the formula or rule
- Values of all in-scope formulas and rules
- Values of all in-scope CSL standard inputs (as well as values of all the inputs' fields, and so on, recursively)

Following are some tips on using the debugger:

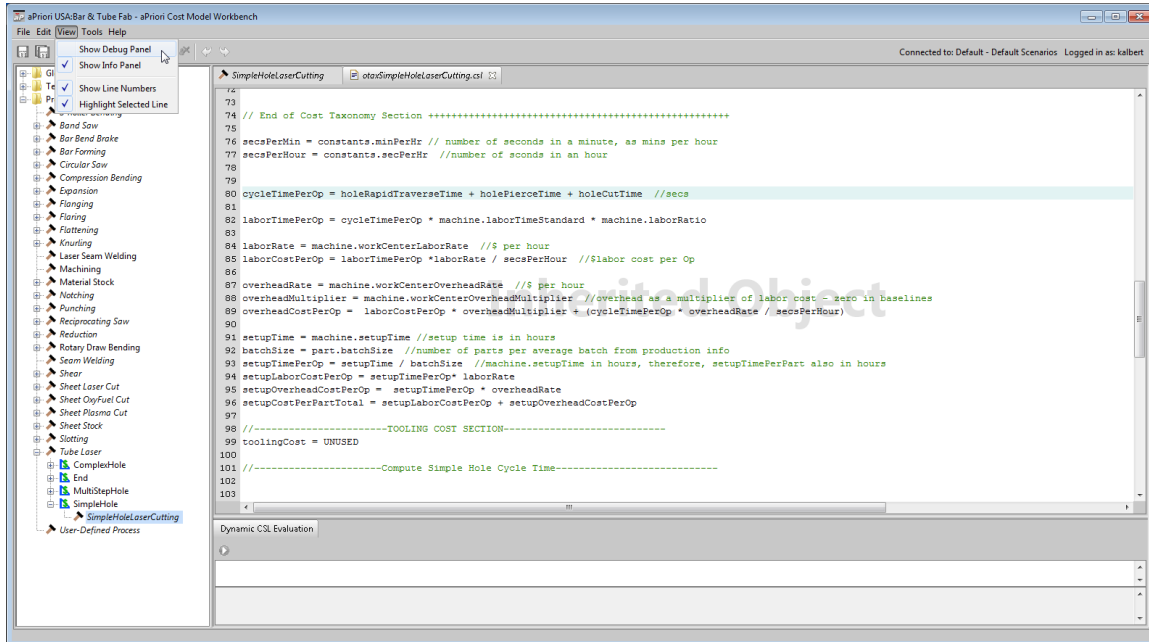
- Save your changes before you run the debugger—otherwise you might not be evaluating the code you expect. The last saved code is always the one evaluated.
- Ensure that the part you have open in aPriori Desktop is using the VPE and process group you currently have open in the debugger
- When you select **Show Debug Panel** from the **View** menu, the debug panel may sometimes be collapsed to a very narrow pane on the right-hand side of the CMWB—just drag the edge to make it bigger.

- Before returning to work with the current part in aPriori, remember to Click the stop button, .

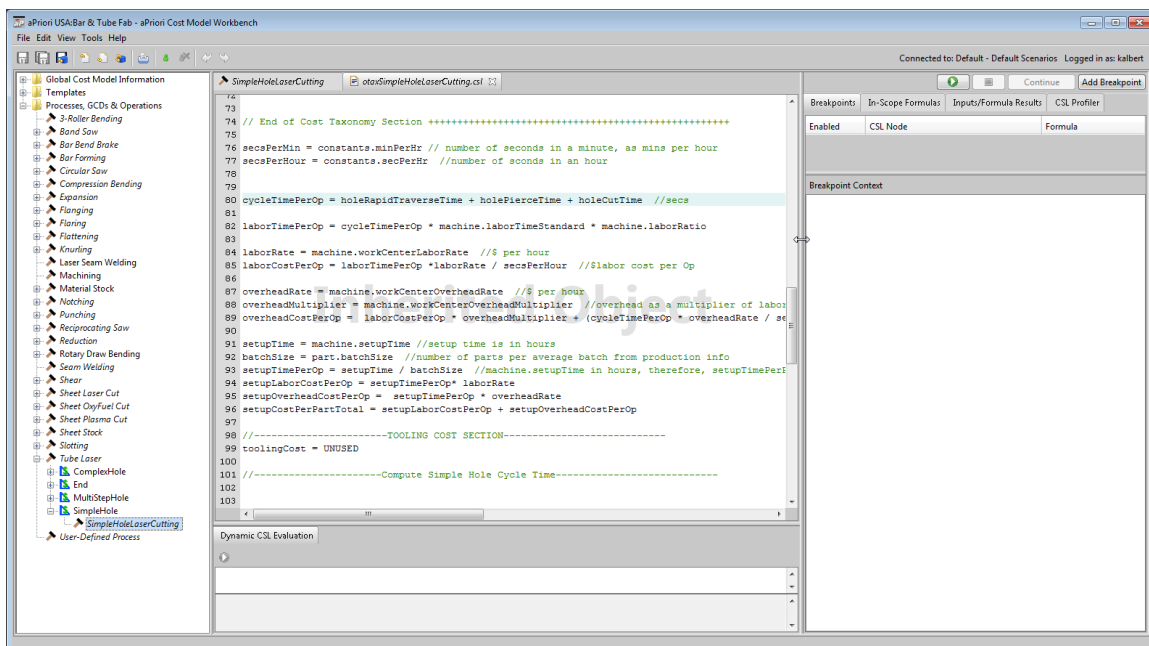
The debug panel also includes a profiler that allows you to determine the proportion of costing time spent on the evaluation of each module, formula, and rule.

Displaying the Debugger

The debug pane is located to the right of the editing pane, and is visible by default. To display it if it is not visible, select **Show Debug Panel** from the CMWB View menu.

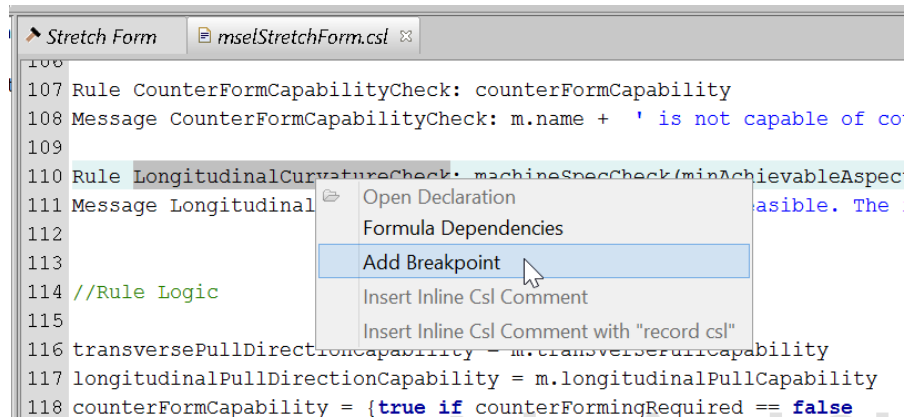


For releases prior to 2018 R1 SP1, you might need to drag the edge of the editing pane to the left to display the debug pane.



Adding Breakpoints

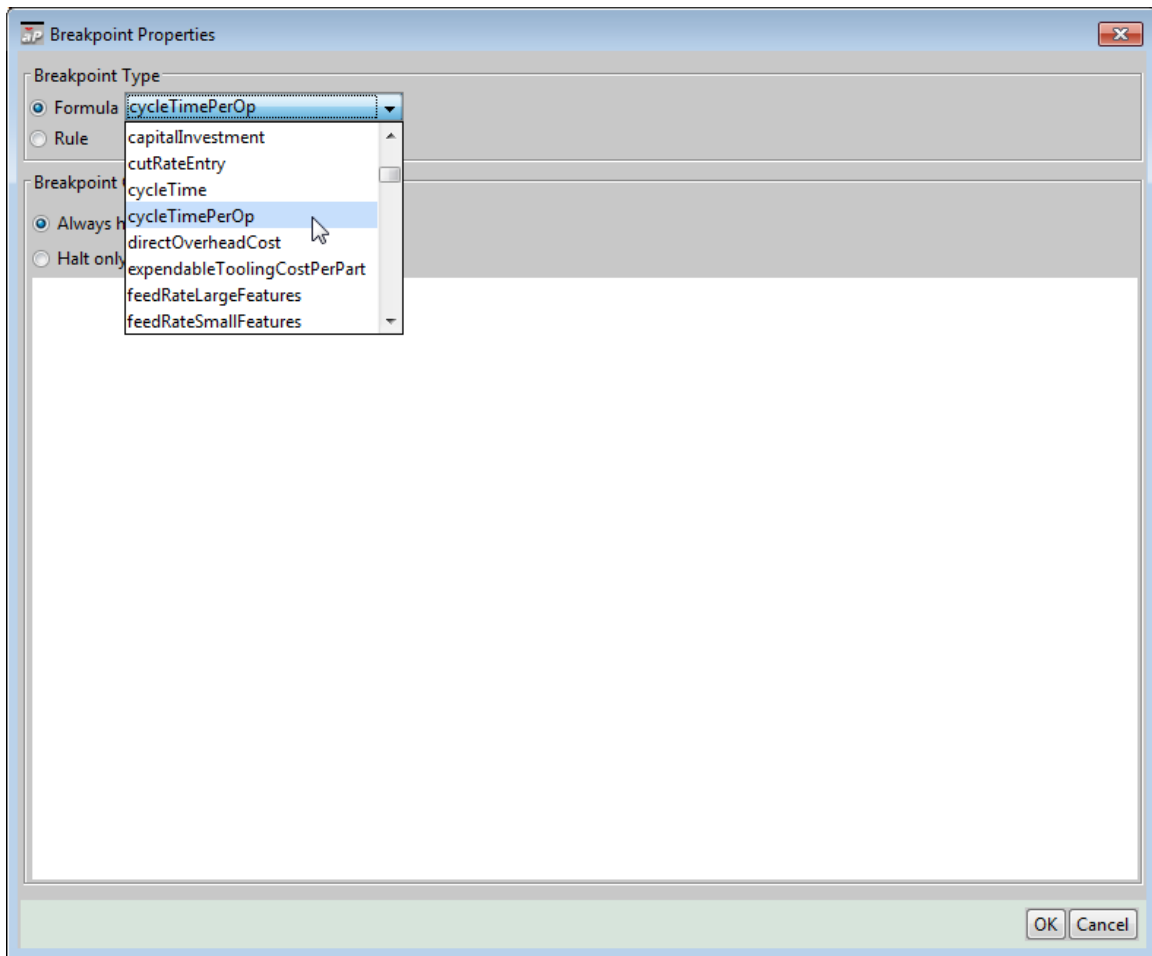
Add a breakpoint as follows: in the editing pane, right click a formula name or rule name and select **Add Breakpoint** from the context menu.



Alternatively, ensure that the debugger is displayed, and follow the steps below. These steps allow you to add a conditional breakpoint, which stops execution at a specified location only when a specified condition is met:

- 1 Click **Add Breakpoint** to display the **Breakpoint Properties** window.

If this button is disabled, open the CSL file that contains the formula for which you want to set the breakpoint.



- 2 Select the breakpoint type and a formula or rule.

Select **Formula** to select a CSL formula from the drop-down list. Select **Rule** to select a rule from the drop-down list.

- 3 Select the breakpoint condition.

Select **Always halt** to stop cost modeling as soon as the formula or rule finishes. Select **Halt only if** to specify an additional condition under which costing is suspended. For example, you might suspend costing only when a certain boundary condition obtains, for example when `gcd.name == 'SimpleHole:1'`.

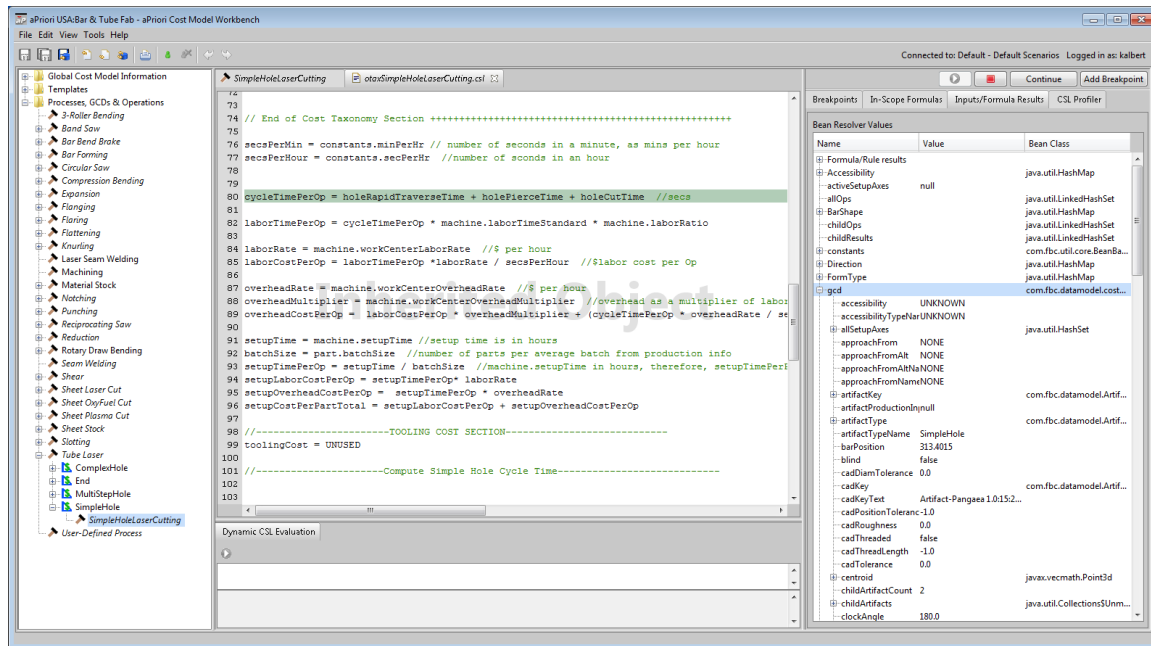
- 4 Click **OK** to close the window and display the breakpoint on the **Breakpoints** tab of the debug pane.

To remove a breakpoint, right-click on it in the **Breakpoints** tab, and select **Remove** from the context menu.

Running the Debugger

Click  to perform costing until the cost engine evaluates the specified formula or rule.

If the CSL file containing the breakpoint is open, the line containing the formula is highlighted and displayed.



Click **Continue** to resume costing until the cost engine reaches the next breakpoint. Note that a single breakpoint can be hit many times in a single cost event. Consider, for example, an operation for a given GCD type, such as SimpleHole. Since the operation may be performed multiple times (once for each GCD of that type), a breakpoint in such an operation may be hit multiple times.

Click  to stop the debugger.

When costing is suspended at a breakpoint, the debugger displays information in the following tabs:

- **Breakpoints:** current breakpoints and context of evaluation of the formula or rule. This tab displays the evaluation stack, including the following:
 - Other formulas or rules whose evaluation required evaluation of the breakpoint
 - Current process or operation
 - Current operation sequence (if any)
 - Current material stock (if any)
 - Current process routing
 - Current GCD
- **In-scope Formulas:** values of all in-scope formulas and rules
- **Inputs/Formula Results:** values of all in-scope CSL standard inputs (as well as values of all the inputs' fields, and so on, recursively). Expand nodes of the tree to see field values. Alternatively, enter a search string in the search field and click **Find**:

The screenshot shows the search interface with the following components:

- Search String:** The text 'len' is entered in the search field.
- Search Controls:** Checkboxes for 'Fields', 'Property', 'Value', and 'Path' are visible. 'Fields' is checked.
- Search Results:** A table with three rows:

Property	Value	Path
length	170.0677	part.length
length	2438.4	stock.length
length	1000.0	dieShoeSizes.dieShoe...


Check any combination of the checkboxes to indicate the search domain:

- **Property:** search for occurrences of a string in the Name column of the tree.
- **Value:** search for occurrences of a string in the Value column of the tree.
- **Path:** search for occurrences of a string in the full path to an input's field (the full path to the **length** field of the input **stock**, for example, is **stock.length**).

You can specify multiple, space-separated search strings; tree nodes are matched only if they contain *all* the search strings. To search for a string that contains spaces (this only applies to field *values*), enclose the whole string in double-quotes.

Click on a search result to highlight the result in the tree. Search results are initially sorted by the level of the tree in which the result occurs (users can re-sort by clicking a column heading).

The search covers the number of levels of the tree specified in the **Depth** field. This is set to 5, initially, but you can decrease the depth to 1, or increase it to 10. In many cases, the greater the depth at which a result appears, the less relevant it is likely to be. (System administrators can configure the maximum available depth by setting the property `ide.display.inputs.max.search.depth` in `apriori.properties`—see the *System Administration Guide*. Note that search depths significantly higher than 10 run the risk of long search times and, in some cases, exhausting the Java heap.)

You can enter a CSL expression in the **Dynamic CSL Evaluation** pane, below the editing pane. Click  in the evaluation pane (or type Ctrl + Enter) to display the result of evaluating, in the current context, the expression you entered.

You can also drag an item from the **Inputs/Formula Results** tree into either an open CSL module or the **Dynamic CSL Evaluation** pane to get a full path reference to the item. See [Dragging Items from the Inputs/Formula Results Tree](#) for more information.

Note that you can undo and redo actions performed in the **Dynamic CSL Evaluation** pane (in the usual way, by selecting **Undo** or **Redo** from Edit menu or by typing Ctrl+Z or Ctrl+Y, when the focus is on the evaluation pane).

Dragging Items from the Inputs/Formula Results Tree

You can drag an item from the **Inputs/Formula Results** tree into either an open CSL module or the **Dynamic CSL Evaluation** pane to get a full path reference to the item, in the current debugger context (see [Running the Debugger](#)).

The screenshot shows the Cost Model Workbench interface. On the left, the **Dynamic CSL Evaluation** pane displays the following code:

```

25 COSTS PER PART
26 *****/
27 // Direct Costs
28 materialCost = GetMaterialCost_SurfaceTreati
29 laborCost = GetLaborCost_SurfaceTreatment_V

```

Below the code, the **Dynamic CSL Evaluation** pane shows the text `part.annualVolumeBean.userValue` entered into the evaluation field. On the right, the **Bean Resolver Values** tree is visible, showing a hierarchy of objects. The `part.annualVolumeBean.userValue` item is highlighted in blue. A red arrow points from this item in the tree to the code in the evaluation pane.

Name	Value
numScrapParts	0.0
op	
part	
active	true
activityInfoEnabled	true
adminInfo	
annualVolume	5500
annualVolumeBean	
cadValue	null
oldValue	null
propertySrc	USER
userValue	5500
value	5500
annualVolumeValue	5500

If you drag an item into the Evaluation pane or a CSL module, and the path to the item does *not* include a collection (that is, if the dragged item has no ancestor in the **Inputs/Formula Results** tree that is a collection), the code that appears is appropriate for use in CSL logic to refer to the dragged item.

However, if the path to the dragged item *does* include a collection, you must modify the code that appears in order to render it appropriate for inclusion in a CSL module. You can modify the code to do any of the following:

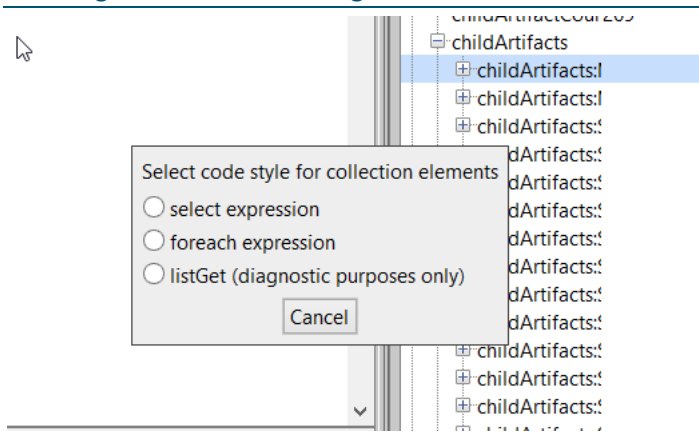
- Refer to the dragged item.
- Refer to a collection of items that are on the same level of the tree as the dragged item and that have the collection as an ancestor.
- Aggregate a collection of values (by taking the sum or maximum, for example) that include items that are on the same level of the tree as the dragged item and that have the collection as an ancestor.

See the following sections for more information:

- [Paths with One Collection](#)
- [Paths with Two Collections](#)
- [Paths with More than Two Collections](#)

Paths with One Collection

If the path to the dragged item includes one collection, the Cost Model Workbench displays the following dialog which lets you choose a code style.



In CSL, collection elements are typically accessed in one of the following ways:

- **select expression:** this is similar to an SQL select statement; it treats the collection as a table with one row for each element and with columns for properties of the elements. See [Query](#). You must modify the select expression in order to focus in on the item or items of interest (see below).
- **foreach expression:** this is a loop that processes each collection element in turn, and builds up a set of results, or formulates a single result. See [Foreach Expressions](#). You must modify the foreach expression in order to focus in on the item or items of interest (see below).
- **listGet call:** a call to `listGet` returns the collection element at a specified position. See `listGet(list, i)`.

Important Note: the `listGet` call is *not* appropriate for use in a CSL module, because the collection element in the path is identified by its *position* in the collection (the second argument to `ListGet`), which is often arbitrary and/or unstable. With select and foreach expressions, in contrast, the collection item or items of interest are generally identified by their property values; the dragged item, for example, might be identified as the diameter of the collection element with the largest **diameter**—see the examples below.

If you choose either **select expression** or **foreach expression**, one or both of the following sorts of modifications is generally needed:

- *Use of a where clause (for select) or conditional (for foreach) to narrow down the selected items, so as to help identify the items of interest.*
- *Use of an aggregation function to further focus in on items of interest, to uniquely identify an item of interest, or to aggregate values of interest.*

See [Paths with One Collection: select expression](#) and [Paths with One Collection: foreach expression](#), below, for more information.

Paths with One Collection: select expression

This section contains an example of modifying the code that appears when you drag an item, so that the code refers to the dragged item. Consider the path to the diameter of the current part's largest-diameter SimpleHole. This path goes from the **part** to the collection of its **childArtifacts** (child GCDs), to the part's largest-diameter **SimpleHole**, to that hole's **diameter**.

Name	Value
childArtifactCount	54
childArtifacts	
childArtifacts:SimpleHole[1]	
accessibility	UNKNOWN
accessibilityTypeName	UNKNOWN
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
artifactKey	
artifactTypeName	SimpleHole
barPosition	-1.0
blind	false
cadKeyText	GCD2:sh (f 10704..
centroid	
childArtifactCount	4
childArtifacts	
circularity	null
clockAngle	-1.0
concentricity	null
coneAngle	90.0
countersinkDirection	UNKNOWN
countersinkDirectionName	UNKNOWN
cylindricity	null
diameter	5.0
diamTolerance	7.0

Suppose you drag this **diameter** item to the Evaluation pane, then choose **select expression** from the **Select code style** dialog. The Evaluation pane is populated with a select expression that requires changes. If you evaluate the unmodified expression, an exception results because some of the selected collection elements lack a **diameter** property (that is, some child GCDs of the part, such as PlanarFaces, don't have a **diameter** property).

Dynamic CSL Evaluation Formula Dependencies

Type Ctrl+Space for auto-complete suggestions

```
select x.diameter from part.childArtifacts x
```

com.fbc.util.core.PropertyDoesNotExistException: Property diameter does not exist in diameter

Bean Resolver Values

Name	Value
childArtifactCount	54
childArtifacts	
childArtifacts:SimpleHole[1]	
accessibility	UNKNOWN
accessibilityTypeName	UNKNOWN
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
artifactKey	
artifactTypeName	SimpleHole
barPosition	-1.0
blind	false
cadKeyText	GCD2:sh (f 10704..
centroid	
childArtifactCount	4
childArtifacts	
circularity	null
clockAngle	-1.0
concentricity	null
coneAngle	90.0
countersinkDirection	UNKNOWN
countersinkDirectionName	UNKNOWN
cylindricity	null
diameter	5.0
diamTolerance	7.0

The expression requires two changes which effectively single out the collection element of interest; that is, the collection element in the path to the dragged object:

- 1 **Addition of a where clause:** since the element of interest is a SimpleHole, the **where** clause must filter the selected objects to include only SimpleHole GCDs. This

change eliminates the cause of the exception, since all SimpleHole GCDs have a **diameter** property. The modified expression evaluates to a collection whose elements are the diameters of the part's SimpleHoles.

The screenshot shows the Dynamic CSL Evaluation interface. The query is: `select x.diameter from part.childArtifacts x where isSimpleHole(x)`. The results pane shows a collection of diameters: `[5.0, 2.5, 2.5]`. The Bean Resolver Values pane shows the structure of the data, with the `diameter` property highlighted in red.

Name	Value
childArtifactCount	54
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility UNKNOWN accessibilityTypeName UNKNOWN approachFrom NONE approachFromAlt NONE approachFromAltName NONE approachFromName NONE artifactKey artifactTypeName SimpleHole barPosition -1.0 blind false cadKeyText GCD2:sh (f 10704... centroid childArtifactCount 4 childArtifacts circularity null clockAngle -1.0 concentricity null coneAngle 90.0 countersinkDirection UNKNOWN countersinkDirectionName UNKNOWN cylindricity null diameter 5.0 diamTolerance 7.0

- 2 *Addition of an aggregation function:* with the where clause, the select expression evaluates to a collection of diameters rather than just the largest diameter. You can single out the largest diameter by using the query-aggregation function **max**:

The screenshot shows the Dynamic CSL Evaluation interface. The query is: `select max(x.diameter) from part.childArtifacts x where isSimpleHole(x)`. The results pane shows a single value: `5.0`. The Bean Resolver Values pane shows the structure of the data, with the `diameter` property highlighted in red.

Name	Value
childArtifactCount	54
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility UNKNOWN accessibilityTypeName UNKNOWN approachFrom NONE approachFromAlt NONE approachFromAltName NONE approachFromName NONE artifactKey artifactTypeName SimpleHole barPosition -1.0 blind false cadKeyText GCD2:sh (f 10704... centroid childArtifactCount 4 childArtifacts circularity null clockAngle -1.0 concentricity null coneAngle 90.0 countersinkDirection UNKNOWN countersinkDirectionName UNKNOWN cylindricity null diameter 5.0 diamTolerance 7.0

Paths with One Collection: foreach expression

If you choose **foreach expression** instead of **select expression** from the **Select code style** dialog, the Evaluation pane or CSL module is populated with a foreach expression. This expression may require changes that focus in on, and possibly aggregate, items of interest. If you evaluate the unmodified expression, an exception may result, as with the select expression (described above).

Dynamic CSL Evaluation Formula Dependencies

Type Ctrl+Space for auto-complete suggestions

```
foreach (x : part.childArtifacts) getAll(var) {
  var = x.diameter
}
```

com.fbc.util.core.PropertyDoesNotExistException: Property diameter does not exist in diameter in CSL module <Unknown Script> at line 2

Bean Resolver Values

Name	Value
childArtifactCount	54
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility UNKNOWN accessibilityTypeName UNKNOWN approachFrom NONE approachFromAlt NONE approachFromAltName NONE approachFromName NONE artifactKey artifactTypeName SimpleHole barPosition -1.0 blind false cadKeyText GCD2:sh (f 10704... centroid childArtifactCount 4 childArtifacts <ul style="list-style-type: none"> circularity null clockAngle -1.0 concentricity null coneAngle 90.0 countersinkDirection UNKNOWN countersinkDirectionName UNKNOWN cylindricity null diameter 5.0 diamTolerance 7.0

One or both of the following sorts of changes are generally required:

- 1 *Conditional assignment of the iteration variable:* the **foreach** loop must filter the results to include only the specific objects or values of interest (diameters of SimpleHoles, in the example).

Dynamic CSL Evaluation Formula Dependencies

Type Ctrl+Space for auto-complete suggestions

```
foreach (x : part.childArtifacts) getAll(var) {
  var = {x.diameter if isSimpleHole(x)
        null otherwise}
}
```

[5.0, 2.5, 2.5]

Bean Resolver Values

Name	Value
childArtifactCount	54
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility UNKNOWN accessibilityTypeName UNKNOWN approachFrom NONE approachFromAlt NONE approachFromAltName NONE approachFromName NONE artifactKey artifactTypeName SimpleHole barPosition -1.0 blind false cadKeyText GCD2:sh (f 10704... centroid childArtifactCount 4 childArtifacts <ul style="list-style-type: none"> circularity null clockAngle -1.0 concentricity null coneAngle 90.0 countersinkDirection UNKNOWN countersinkDirectionName UNKNOWN cylindricity null diameter 5.0 diamTolerance 7.0

- 2 *Replacement of the aggregation function:* the unmodified foreach expression uses the foreach-aggregation function **getAll**. You can further narrow or aggregate the foreach results by using a different aggregation function. This example singles out the largest diameter by using the foreach-aggregation function **getMax**:

The screenshot shows the 'Dynamic CSL Evaluation' pane on the left and the 'Bean Resolver Values' pane on the right. The CSL code in the left pane is:

```
foreach (x : part.childArtifacts) getMax(var) {
  var = {x.diameter if isSimpleHole(x)
        null otherwise}
}
```

The 'Bean Resolver Values' pane on the right shows a tree structure of values. A red box highlights the 'childArtifactsSimpleHole[1]' node, which contains the following properties and values:

Name	Value
accessibility	UNKNOWN
accessibilityTypeName	UNKNOWN
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
artifactKey	
artifactTypeName	SimpleHole
barPosition	-1.0
blind	false
cadKeyText	GCD2:sh (f 10704...
centroid	
childArtifactCount	4
childArtifacts	
circularity	null
clockAngle	-1.0
concentricity	null
coneAngle	90.0
countersinkDirection	UNKNOWN
countersinkDirectionName	UNKNOWN
cylindricity	null
diameter	5.0
diamTolerance	7.0

A red arrow points from the 'diameter' property in the Bean Resolver Values pane to the 'x.diameter' property in the CSL code. The value '5.0' is also visible in a small box at the bottom left of the CSL pane.

Paths with Two Collections

If you drag an item to the Evaluation pane or a CSL module, and the path to the dragged item includes exactly two collections, the Cost Model Workbench displays the following dialog:

The dialog box titled 'Select code style for collection elements' contains three radio button options:

- foreach wrapping a select
- select from right-most collection
- listGet (diagnostic purposes only)

A 'Cancel' button is located at the bottom of the dialog. To the right of the dialog, a tree view shows a path of nodes: app, app, app, artif, artif, barf, blin, cadl, cent, chik.

If you choose **foreach wrapping select**, the Evaluation pane or CSL module is populated with a foreach expression whose body contains a select expression. This code requires changes in order to aggregate or focus in on items of interest.

Suppose, for example, you want to construct a CSL expression that evaluates to the surface area of the part's largest-surface-area SimpleHole. To start, you might drag to the Evaluation pane the **finishArea** (surface area) item of a CurvedWall child of one of the part's SimpleHoles. (SimpleHoles have no surface area property, so the code we construct will reference the hole's CurvedWall children.) This path goes from the **part** to the collection of its **childArtifacts**, to the SimpleHole, to the collection of that SimpleHole's **childArtifacts**, to the CurvedWall, to the **areaFinished** of that CurvedWall.

Now suppose that, after dragging the **finishArea** item, you choose **foreach wrapping select** from the **Select code style** dialog. The Evaluation pane is populated with an initial CSL expression based on the dragged item. In most cases, evaluating the unmodified expression will result in an exception due to certain collection elements visited by the select statement lacking an **areaFinished** property.

The screenshot shows the Dynamic CSL Evaluation window with the following code:

```
foreach (x : part.childArtifacts) getAll(var) {
  var = select y.areaFinished from x.childArtifacts y
}
```

An error message is displayed at the bottom: `com.fbc.util.core.PropertyDoesNotExistException: Property areaFinished does not exist in areaFinished in CSL module <Unknown Script> at line 2`. A red arrow points from the error message to the `areaFinished` property in the Bean Resolver Values table.

Name	Value
childArtifactCount	70
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility accessibilityTypeName approachFrom approachFromAlt approachFromAltName approachFromName artifactKey artifactTypeName barPosition blind cadKeyText centroid childArtifactCount childArtifacts <ul style="list-style-type: none"> childArtifacts:CurvedWall[1] <ul style="list-style-type: none"> approachFrom approachFromAlt approachFromAltName approachFromName areaFinished
areaFinished	50.2655

The following changes are required to render this code appropriate for use in a CSL module:

- 1 **Addition of a where clause:** the first collection in the path is that containing the part's top-level GCDs. It includes the SimpleHole that appears within path to the dragged item. From this collection, the path to the dragged item includes a CurvedWall, so we use a **where** clause to filter the objects selected from this collection so as to include only CurvedWall GCDs. This change eliminates the cause of the exception since all CurvedWall GCDs have an **areaFinished** property. The modified expression evaluates to a collection whose elements are collections of **areaFinished** values. There is one collection of **areaFinished** values for each of the part's children that itself has CurvedWall children. Each sub-collection contains the **areaFinished** values for the CurvedWall children.

The screenshot shows the Dynamic CSL Evaluation window with the following code:

```
foreach (x : part.childArtifacts) getAll(var) {
  var = select y.areaFinished from x.childArtifacts y where isCurvedWall(y)
}
```

The error message is gone. A red arrow points from the `areaFinished` property in the Bean Resolver Values table to the `areaFinished` property in the code.

Name	Value
childArtifactCount	70
childArtifacts	<ul style="list-style-type: none"> childArtifacts:SimpleHole[1] <ul style="list-style-type: none"> accessibility accessibilityTypeName approachFrom approachFromAlt approachFromAltName approachFromName artifactKey artifactTypeName barPosition blind cadKeyText centroid childArtifactCount childArtifacts <ul style="list-style-type: none"> childArtifacts:CurvedWall[1] <ul style="list-style-type: none"> approachFrom approachFromAlt approachFromAltName approachFromName areaFinished
areaFinished	50.2655

- 2 **Conditional assignment of the iteration variable:** we now have **areaFinished** values for CurvedWall GCDs, but some of these may belong to top-level GCDs of types other than SimpleHole. We stated earlier we were only interested in SimpleHoles, so we need to filter the collection of top-level GCDs to remove non-SimpleHole types. We use conditional assignment of the foreach expression's iteration variable to achieve this:

Dynamic CSL Evaluation Formula Dependencies

```

foreach (x : part.childArtifacts) getAll(var) {
  var = {select y.areaFinished from x.childArtifacts y where isCurvedWall(y) _
        if isSimpleHole(x)
        null otherwise}
}

```

Type Ctrl+Space for auto-complete suggestions

Bean Resolver Values

Name	Value
-childArtifactCount	70
childArtifacts	
childArtifacts:SimpleHole[1]	
accessibility	UNKNOWN
accessibilityTypeName	UNKNOWN
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
artifactKey	
artifactTypeName	SimpleHole
barPosition	-1.0
blind	false
cadKeyText	GCD2:sh (f 216 20...
centroid	
childArtifactCount	4
childArtifacts	
childArtifacts:CurvedWall[1]	
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
areaFinished	50.2655

[[50.2655, 50.2655], [50.2655, 50.2655], [31.4159, 31.4159]]

- 3 *Addition of a query-aggregation function:* with the where clause and foreach conditional assignment, the expression evaluates to a collection of pairs (since each SimpleHole has exactly two CurvedWall children). You can obtain the total surface area for each SimpleHole by introducing the aggregation function **sum** to the select expression. In effect, this adds together the two **areaFinished** values in each pair:

Dynamic CSL Evaluation Formula Dependencies

```

foreach (x : part.childArtifacts) getAll(var) {
  var = {select sum(y.areaFinished) from x.childArtifacts y where isCurvedWall(y) _
        if isSimpleHole(x)
        null otherwise}
}

```

Type Ctrl+Space for auto-complete suggestions

Bean Resolver Values

Name	Value
-childArtifactCount	70
childArtifacts	
childArtifacts:SimpleHole[1]	
accessibility	UNKNOWN
accessibilityTypeName	UNKNOWN
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
artifactKey	
artifactTypeName	SimpleHole
barPosition	-1.0
blind	false
cadKeyText	GCD2:sh (f...
centroid	
childArtifactCount	4
childArtifacts	
childArtifacts:CurvedWall[1]	
approachFrom	NONE
approachFromAlt	NONE
approachFromAltName	NONE
approachFromName	NONE
areaFinished	50.2655

[100.531, 100.531, 62.8318]

- 4 *Addition of a foreach-aggregation function:* with the use of the query-aggregation function, the expression evaluates to a collection of surface area values. We can now single out the largest of these values with the foreach-aggregation function **getMax**:

The screenshot shows the 'Dynamic CSL Evaluation' window. The left pane contains the following code:

```
foreach (x : part.childArtifacts) getMax(var) {
  var = {select sum(y.areaFinished) from x.childArtifacts y where isCurvedWall(y)
        if isSimpleHole(x)
        null otherwise}
}
```

The right pane shows the 'Bean Resolver Values' tree. A red arrow points from the 'select from' expression in the code to the 'areaFinished' property in the tree, which has a value of 50.2655.

If, instead of **foreach wrapping select**, you choose **select from right-most collection**, the code that appears uses a select expression that queries elements from the collection that is nearest the dragged item, i.e. the right-most collection in the path to the dragged item. The generated expression includes calls to **listGet**, which must be replaced in order to render the code appropriate for use in CSL module.

If you choose **listGet**, the code that appears is not suitable for inclusion in a CSL module, but rather is for diagnostic purposes only.

Paths with More than Two Collections

If you drag an item to the Evaluation pane or a CSL module, and the path to the dragged item includes more than two collections, the Cost Model Workbench displays the following dialog:

Select code style for collection elements

select from right-most collection

listGet (diagnostic purposes only)

If you choose **select from right-most collection**, the code that appears uses a select expression that queries elements from the collection that is nearest the dragged item, i.e. the right-most collection in the path to the dragged item. The generated expression includes calls to **listGet**, which must be replaced in order to render the code appropriate for use in CSL module.

If you choose **listGet**, the code that appears is not suitable for inclusion in a CSL module, but rather is for diagnostic purposes only.

CSL Reference Information

For detailed information on the CSL language, see [Cost Scripting Language Reference](#). That chapter also covers advanced constructs not discussed here, including set blocks, foreach expressions, and associative collection access.

Viewing and Editing CSL Modules

For a given process group, each CSL module is either global to the process group or associated with a particular node of one of the group's process or operation templates. Recall that each node represents a process or operation (or else is a branch node—see [Working with](#)).

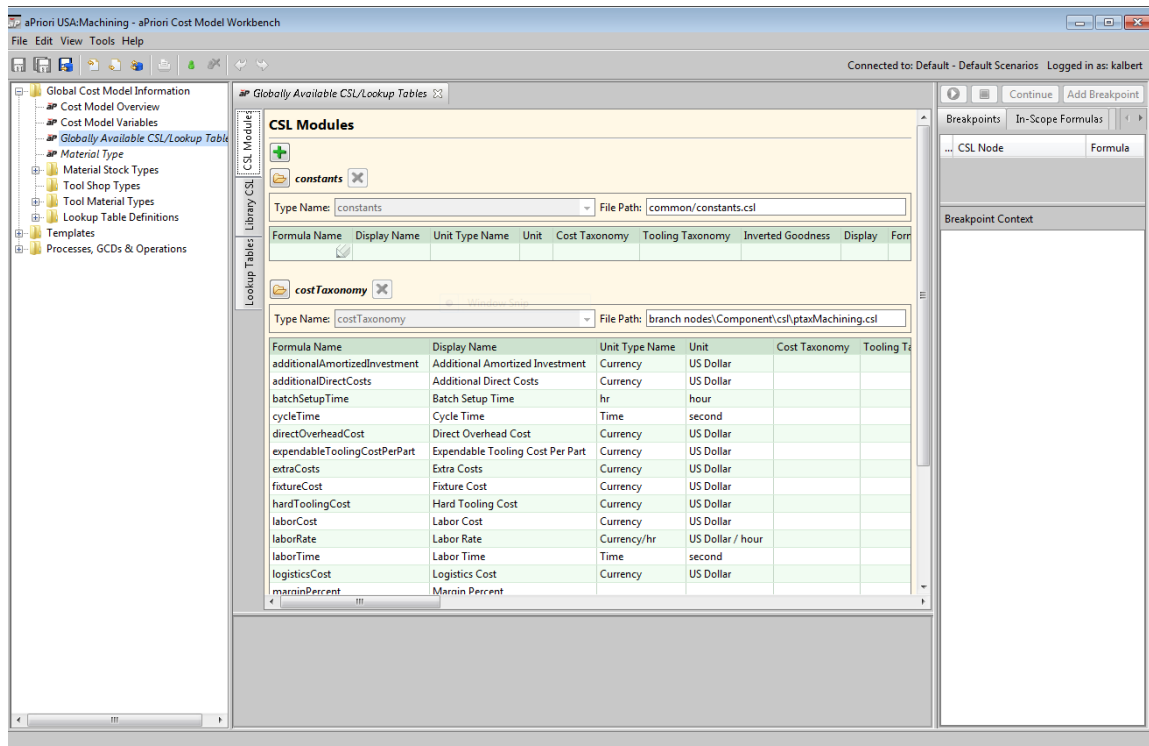
This section covers the following topics:

- Navigating to Global CSL Modules
- Navigating to the CSL Modules for a Given Node
- Viewing CSL Modules
- Editing CSL Modules
- Adding CSL Modules
- Deleting CSL Modules

Navigating to Global CSL Modules

Follow these steps to display information about and links to the global CSL modules for the current process group:

- 1 In the CMWB navigation pane, expand the **Global Cost Model Information** node.
- 2 Double click **Globally Available CSL/Lookup Tables** in the navigation pane.
- 3 In the editing pane, select the **Library CSL** tab (for library modules) or the **CSL Modules** tab (for non-library modules).



For each global CSL module, the editing pane displays the following:

- Module name
- Module type name
- File path for the module
- Table of output formulas
- Folder icon for module viewing or editing
- X symbol for module deletion

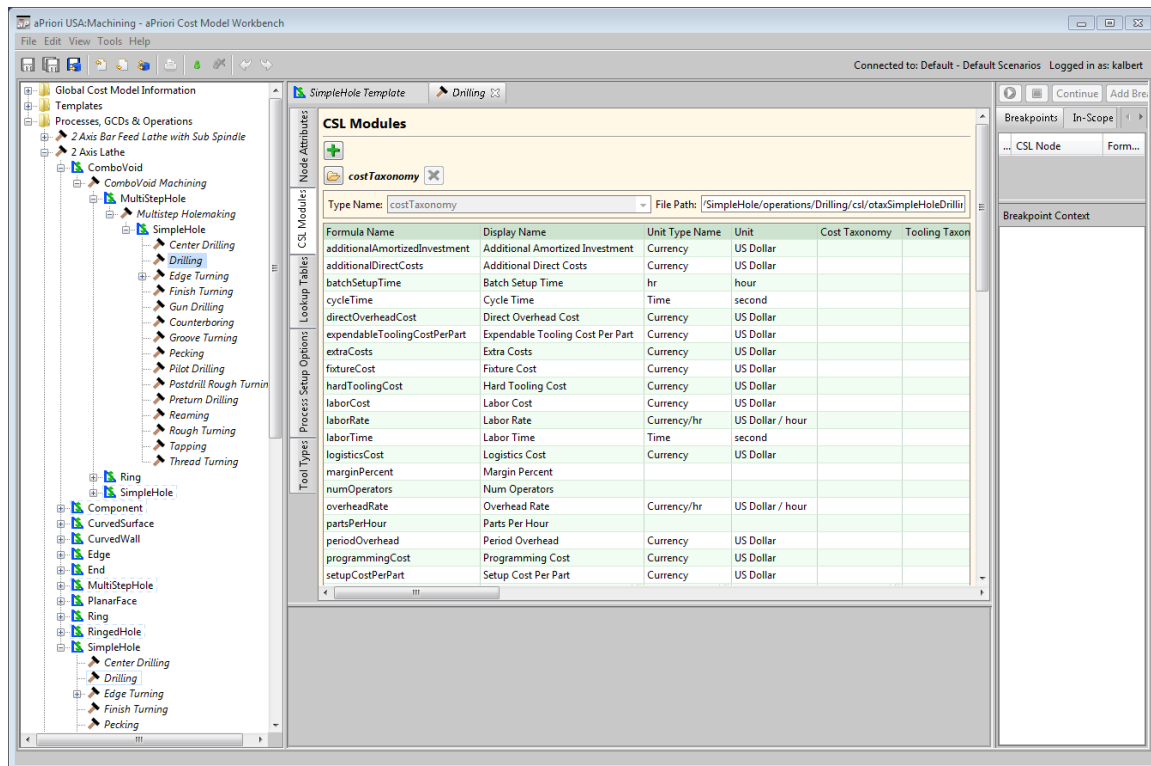
The editing pane also displays a + icon for adding a module. See [Adding CSL Modules](#).

Navigating to the CSL Modules for a Given Node

Besides the global CSL modules (see [Navigating to Global CSL Modules](#)), each CSL module is associated with a single template node. (Note that same template node, in this sense of node, can appear in multiple templates.) There are two general ways to access the CSL modules associated with a given template node: directly from the navigation pane, and from the template graph below the editing pane.

For a node that represents a process or operation (as opposed to a branch node—see [Working with](#)), you can access the node's modules directly from the navigation pane as follows:

- 1 In the navigation pane, expand **Processes, GCDs & Operations**.
- 2 If the node represents a process, double click the process that the node represents. Otherwise, expand a process that can serve as an ancestor (in the process-operation hierarchy—see [Cost Engine Details](#)) of the desired operation.
- 3 Expand a GCD type to whose creation the desired operation can contribute.
- 4 Double click the desired operation under the expanded GCD, if it appears. Otherwise, expand an operation that can serve as ancestor (in the process/operation hierarchy) of the desired operation, and go back to step 3 to continue down the hierarchy.
- 5 In the editing pane, select the **CSL Modules** tab.



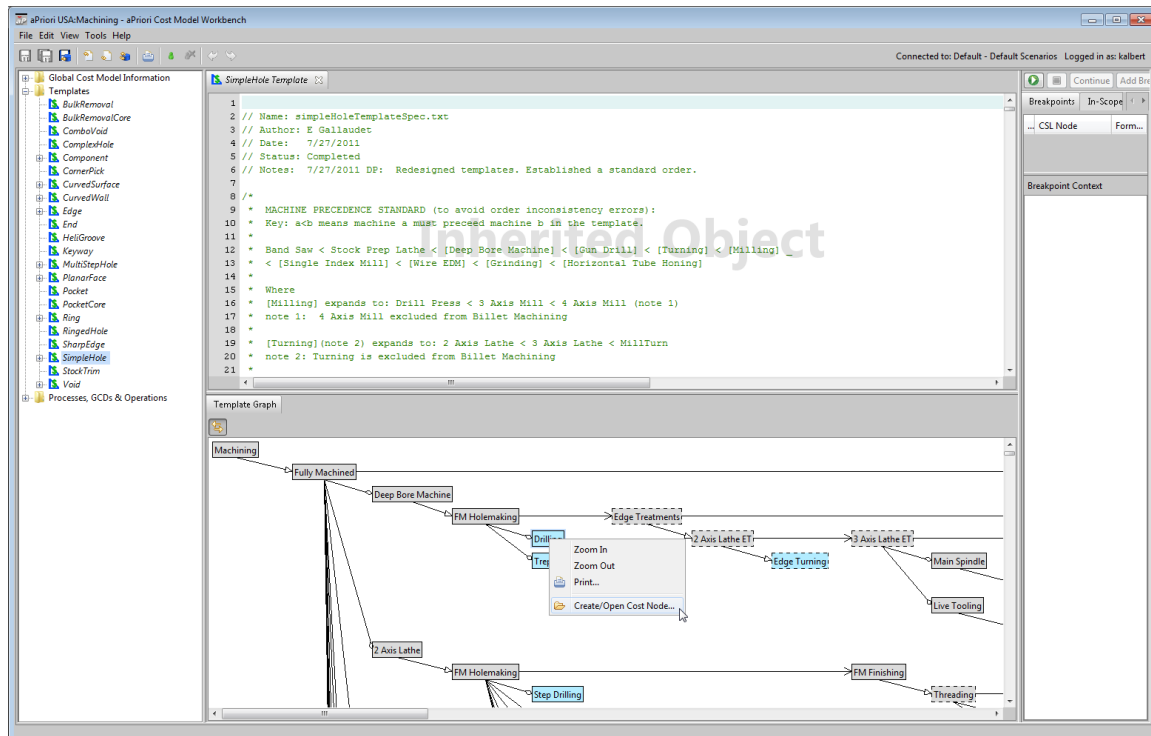
For branch nodes (see [Working with](#)), you can access a module for a given node directly from the navigation pane as follows:

- 1 In the navigation pane, expand **Templates**.
- 2 Expand a GCD type in whose template the desired node occurs.
- 3 Double click the desired node under the expanded GCD.
- 4 In the editing pane, select the **CSL Modules** tab.

To access, from a template graph, the modules for a given node, follow these steps:

Caution: Use this navigation method only to navigate to processes, operations, or branch nodes that you are certain exist in the navigation pane. Some template nodes exist only in a template (for organizational purposes) and have no associated data or logic in the cost model. Using this method on such a node unnecessarily creates an empty node in the navigation pane and clutters the cost model.

- 1 In the navigation pane, expand **Templates**.
- 2 Double click a GCD type in whose template the desired node occurs. The template specification appears in the editing pane, and the template graph appears in the pane below it.



3 In the template graph, right click on the desired node, and select **Create/Open Cost Node**.

4 In the editing pane, select the **CSL Modules** tab.

For each CSL module, the editing pane displays the following:

- Module name
- Module type name
- File path for the module
- Table of output formulas
- Folder icon for module viewing or editing
- X symbol for module deletion

The editing pane also displays a + icon for adding a module. See [Adding CSL Modules](#).

Viewing CSL Modules

To view a CSL module, follow these steps:

1 For a global CSL module, navigate to the global CSL modules (see [Navigating to Global CSL Modules](#)). Be sure to select the **Library CSL** tab for library modules and the **CSL Modules** tab for non-library modules.

For CSL modules associated with a node, navigate to the CSL modules for the node—see [Navigating from the Template Graph to the Data for a Given Node](#).

2 Click the folder icon for the desired module. The module text appears in the editing pane.

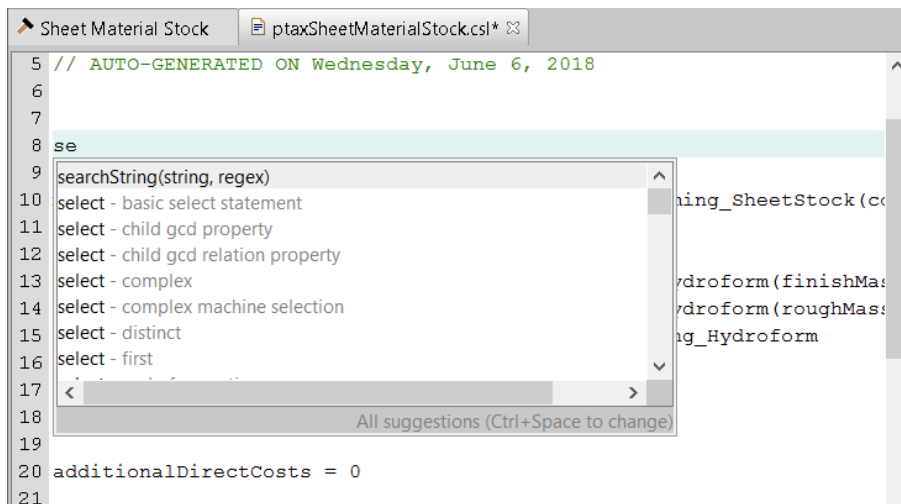
Editing CSL Modules

To edit a CSL module, follow these steps:

- 1 For a global CSL module, navigate to the global CSL modules (see [Navigating to Global CSL Modules](#)). Be sure to select the **Library CSL** tab for library modules and the **CSL Modules** tab for non-library modules.
For CSL modules associated with a node, navigate to the CSL modules for the node— see [Navigating from the Template Graph to the Data for a Given Node](#).
- 2 Click the folder icon for the desired module. The module text appears in the editing pane.
- 3 Select **Override Object** from the **CMWB Edit** menu. You can now modify the text. See also [Using Completion](#), below.
- 4 To add or remove output formulas, select the tab (at the top of the editing pane) for the node associated with the module you are editing. In other words, go back is the screen on which you just clicked the folder icon. Modify the module's formula table. To add a formula, use the blank line at the end of the table. To remove a formula, right click in it and select **Remove**.
- 5 Select **Save All** from the **File** menu to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu.

Using Completion

The CSL editor supports code completion. Type Ctrl and Space together to pop up a list of suggested completions for the word you are typing.



The list of suggestions includes the following:

- CSL predefined (built-in) functions, such as **searchString** and **asList**, whose names include the sequence of letters that you have typed so far.
- CSL templates, such as **select** and **foreach** templates, whose names include the sequence of letters you have typed so far. A CSL template is a canonical

example of a CSL construct (such as a **select** expression). It contains placeholders, which you will replace with expressions of your choice in order to tailor the example to your needs. Note that you can define your own templates in order to supplement the built in, system-supplied templates—see [Defining CSL Code Templates](#).

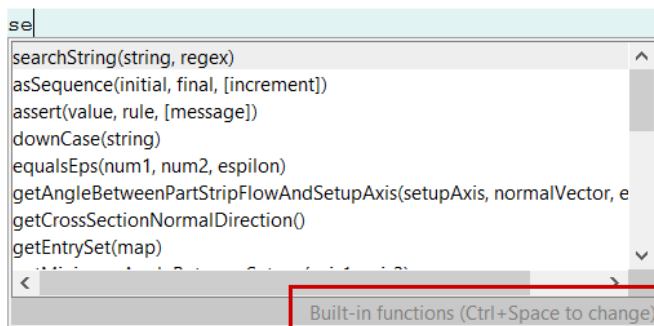
- CSL standard inputs, such as **plant** and **childResults**, whose names include the sequence of letters you have typed so far.

The list begins with suggestions that start with the letters typed so far, arranged alphabetically. These are followed by suggestions that include, but don't start with, the letters typed so far, arranged alphabetically.

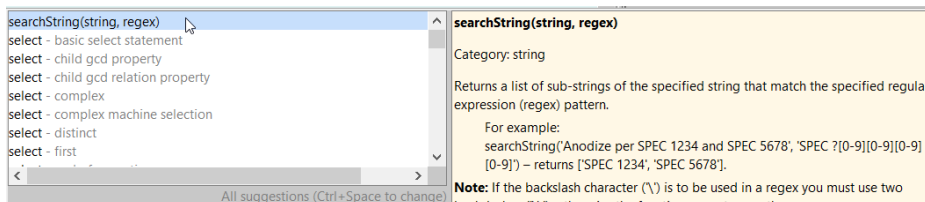
Typing Control and Space pops up a list with all suggestions; typing control space additional times toggles through filtered lists:

- System-supplied template suggestions
- User-defined template suggestions
- Built-in function suggestions
- Standard input suggestions
- All suggestions

The current category of suggestions is displayed in the lower right:



Click on a suggested function or template to display documentation for it.



Double click a suggestion to insert its associated code into the CSL module at the cursor location:

- Functions: double click a function suggestion to insert a template for a call to the function. For functions with arguments, you will have to replace the argument-placeholders with the appropriate expressions.
- Templates: double click a template suggestion to insert a template for a formula or expression. You will have to replace the placeholders with appropriate expressions.
- Inputs: double click an input suggestion to insert it into the code.

You can also use the up/down cursor keys to select an entry (and bring up any associated documentation), and you can press **RETURN/ENTER** to insert the selected entry's CSL.

The following shows the associated code for the function `searchString`, and for the templates `select`, `foreach`, and `with`:

```
searchString(string, regex)

select x from collection x where x.attribute == variable

foreach (variable1: select x from collection x) getAll(variable2) {
    variable2 = select x.attribute from variable.collection x
}

variable = {
    with {
        a = x + y
        b = z
    }
    a + b
}
```

Note that the suggestion list only includes CSL standard inputs when costing is stopped at a CSL breakpoint. The suggestions include only the inputs that are defined or available (that is, in scope) at that point in the cost model execution.

Defining CSL Code Templates

Follow these steps to define a CSL code template:

- 1 Create a file called `user-defined-csl-templates.xml`. Do one of the following:
 - Save the file in the application-data directory of your aPirori installation (for example, in `C:\Users\<user>\AppData\Local\APriori\18.3`).
 - Save the file in a directory of your choice that you can continue to use when you upgrade to new versions of aPirori. Set the property `ide.csl.templates.dir` in `apriori.properties` to the path for this directory (see the *System Administration Guide*).
- 2 Edit the file to include one `templates` element, and within that, as subelements, one `template` element for each template that you want to define. Here is an example:

```
<templates>

<template name="Basic Template" description="first example">
  <documentation><![CDATA[
    Type some documentation here...
  ]]></documentation>
  <csl>basic.single(line).code.fragment</csl>
</template>

<template name="Longer Template" description="another example">
  <documentation><![CDATA[
    Type some documentation here...
  ]]></documentation>
  <csl><![CDATA[
    foreach (variable1: select x from collection x) getSum(variable2) {
```

```

        variable2 = select x.attribute from variable.collection x
    }
  ]]></cs1>
</template>

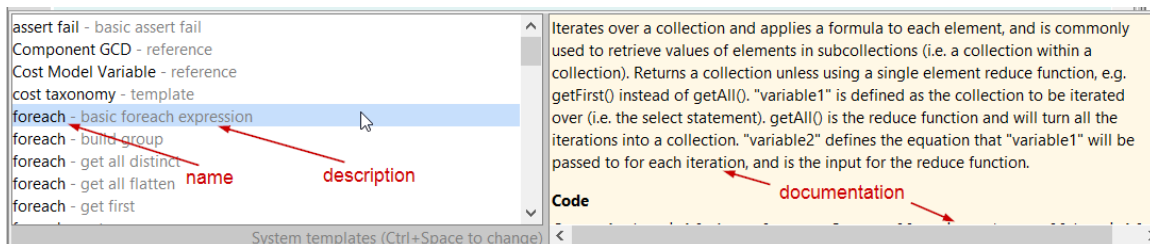
...

</templates>

```

Each **template** element has the following attributes and subelements:

- **name:** template name
- **description** (optional): short description of the template
- **documentation** (optional): documentation for the template
- **cs1:** the template's associated code example



The use of **CDATA** tags is generally recommended for the **documentation** and **cs1** elements, in order to avoid problems arising from the use of XML reserved characters in the element's contents.

Adding CSL Modules

To add a CSL module, follow these steps:

- 1 For a global CSL module, navigate to the global CSL modules (see [Navigating to Global CSL Modules](#)). Be sure to select the **Library CSL** tab for library modules and the **CSL Modules** tab for non-library modules.

For CSL modules associated with a node, navigate to the CSL modules for the node— see [Navigating to the Data for a Process, Operation or Branch Node](#).

- 2 Select **Override Object** from the **CMWB Edit** menu.
- 3 Click the green plus icon at the top or bottom of the editing pane. Information for a new module appears in the editing pane
- 4 For modules other than library modules, select the type of the new module from the dropdown list for the **Type Name** field.

For library modules, enter the name of the new module in the **File Name** field. The name should start with `lib` and end with `.cs1`.

- 5 Enter a line in the formula table for each output formula (if any) of the new module. Standard taxonomy formulas are added automatically.
- 6 Select **Save** from the **File** menu.

- 7 Click the folder icon for the new module. The module content appears in the editing pane. The module is initially empty (except for default formulas that set to 0 certain high-level values, such as `cycleTime` and `pieceCost`).
- 8 After adding the text of the new module, select **Save** from the **File** menu to save your changes.
- 9 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu.

Deleting CSL Modules

To delete a CSL module, follow these steps:

- 1 For a global CSL module, navigate to the global CSL modules (see [Navigating to Global CSL Modules](#)). Be sure to select the **Library CSL** tab for library modules and the **CSL Modules** tab for non-library modules.

For CSL modules associated with a node, navigate to the CSL modules for the node— see [Navigating from the Template Graph to the Data for a Given Node](#).
- 2 Click the folder icon for the desired module. The module text appears in the editing pane.
- 3 Select **Override Object** from the **CMWB Edit** menu.
- 4 At the top of the editing pane, select the tab for the node associated with the module you want to delete. In other words, go back to the screen on which you just clicked the folder icon.
- 5 Click the red X icon to delete the file.
- 6 Select **Save** from the **File** menu to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu.

Creating and Deleting Processes, Operations, and Branch Nodes

This section covers how to create and delete processes, operations, and branch nodes (see [Working with](#)). You can create a new process and operation either from scratch or by copying the modules associated with an existing process or operation.

This section covers the following topics:

- [Creating and Copying Processes](#)
- [Creating and Copying Operations](#)
- [Creating Branch Nodes](#)
- [Deleting Processes, Operations, and Branch Nodes](#)

Creating and Copying Processes

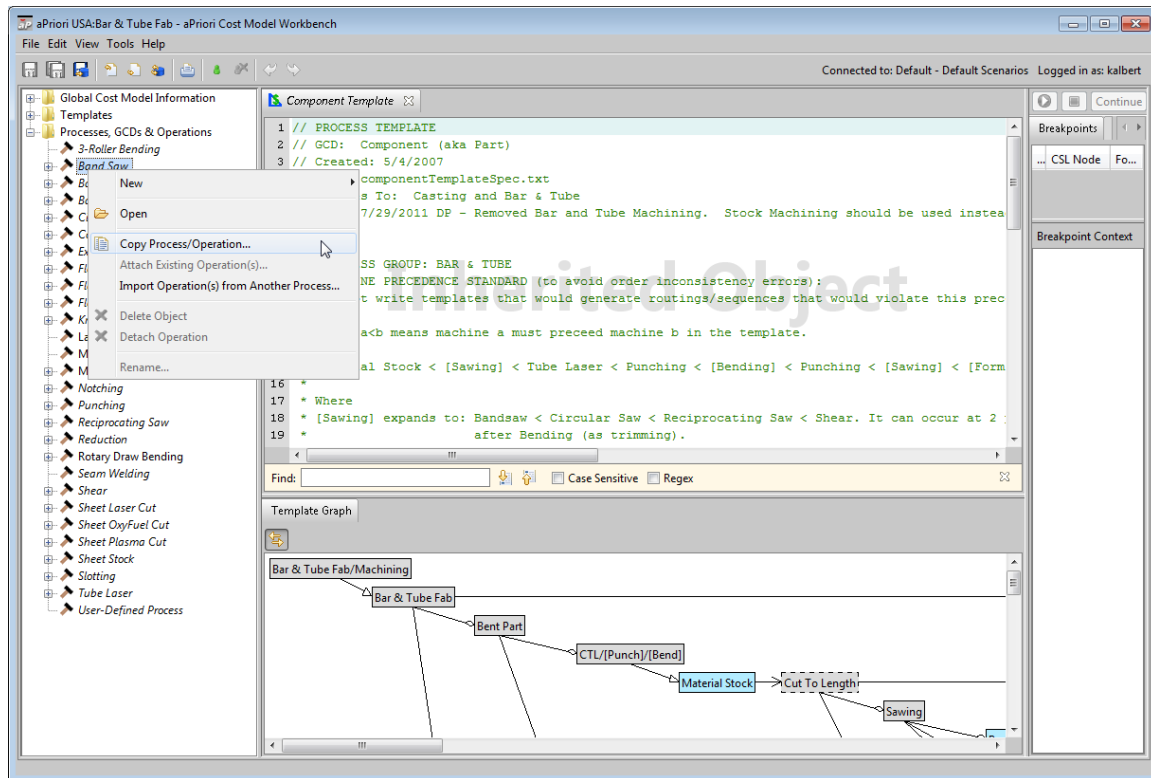
You can create a new process in three different ways:

- By creating a copy of another process. The copy includes copies of the source process's metadata definitions and CSL modules. With this method, you can also optionally copy specified operations. See [Creating a Process by Copying](#).
- By creating a new process with standard taxonomy and machine selection modules. These modules contain standard accounting formulas, and can serve as a starting point for development of the new process's CSL modules. With this method, you can also optionally copy a specified machine type. See [Creating a Process with Standard Taxonomy and Machine Selection Modules](#).
- By creating a new process entirely from scratch. See [Creating a Process from Scratch](#).

Creating a Process by Copying

To create a new process by copying the CSL modules associated with an existing process, follow these steps:

- 1 In the navigation pane, under **Processes, GCDs, & Operations**, right-click on the process you want to copy. Select **Copy Process/Operation...** from the context menu. The **Copy Process** dialog appears.



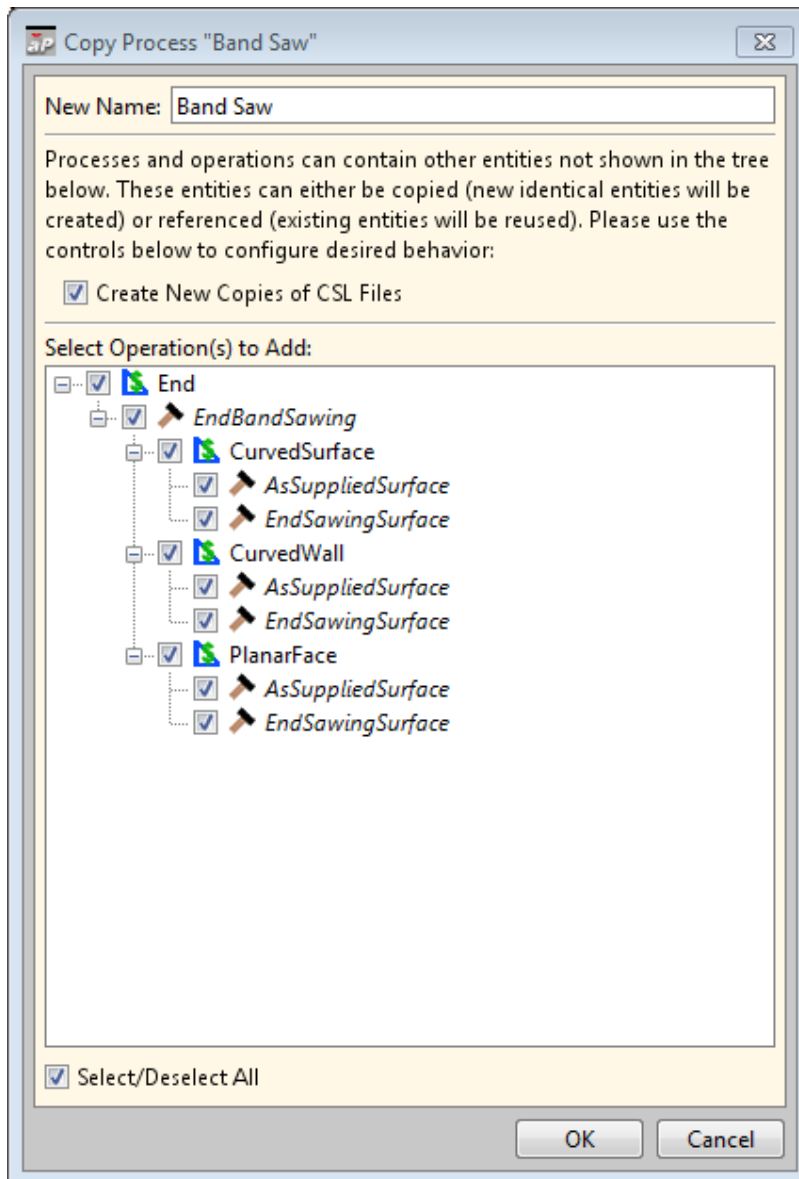
- 2 In the **Copy Process** dialog, modify the following sections, if necessary:

New Name: enter a name for the new process.

Create New Copies of CSL Files: leave this checked if you want to also copy all the modules associated with some of the process's potential descendent operations.

Uncheck it, if you want to use some of the identical potential descendent operation modules.

Select Operation(s) to Add: leave checked only those operations that you want to serve as potential descendents of the new process. Expand the GCD-operation tree, if necessary.



- 3 Modify the template for the component GCD to refer to the new process (and modify templates for new operations, if any)--see [Working with Templates](#). Create or modify the appropriate CSL modules.

Creating a Process with Standard Taxonomy and Machine Selection Modules

This section describes how to create a new process with standard taxonomy and machine selection modules. These modules contain standard accounting formulas, and can serve as a starting point for development of the new process's CSL modules. Here is a portion of the standard CSL from the taxonomy module:

```

import libCommonAccounting_v4.csl

/*
 * Name: ptax
 * Author:
 * Created/Modified:
 * Purpose: Process taxonomy for this process.
 */

// Used for fail messages
machine_name = machine.name

/*****
COSTS PER PART
*****/
// Direct Costs
materialCost = UNUSED
laborCost = GetLaborCost(laborTime, cycleTime, laborRate, finalYield)
directOverheadCost = GetDirectOverheadCost(laborCost, cycleTime, laborTime, overheadMultiplier, overheadRat

// Other Direct Costs
expendableToolingCostPerPart = GetExpendableToolingCostPerPart(finalYield)
logisticsCost = GetLogisticsCost
setupCostPerPart = GetSetupCostPerPart(setupTimePerPart, laborRate, overheadRate)
setupTimePerPart = GetSetupTimePerPart(part.batchSize, cycleTime, batchSetupTime)
additionalDirectCosts = GetAdditionalDirectCosts(finalYield)

// Piece Cost
extraCosts = GetExtraCosts

// Fixed Costs
additionalAmortizedInvestment = GetAdditionalAmortizedInvestment

// TOTAL

// Fully Burdened Cost
periodOverhead = GetPeriodOverhead(periodOverheadCoefficient, laborCost)

```

Here is the standard CSL from the machine selection module:

```

//Comments: This file contains conditional tests that must be true in order to select a given machine.
//Some files may actively select the best machine
//-----

//select the 'default' machine: if default not capable, then select the lowest cost capable machine (overhe

machine = select first(m) from machines m _
           order by isDefaultMachine(m) desc, totalOverheadRate(m) // sorts defaultMachine first

           isDefaultMachine(m) = (m == defaultMachine)

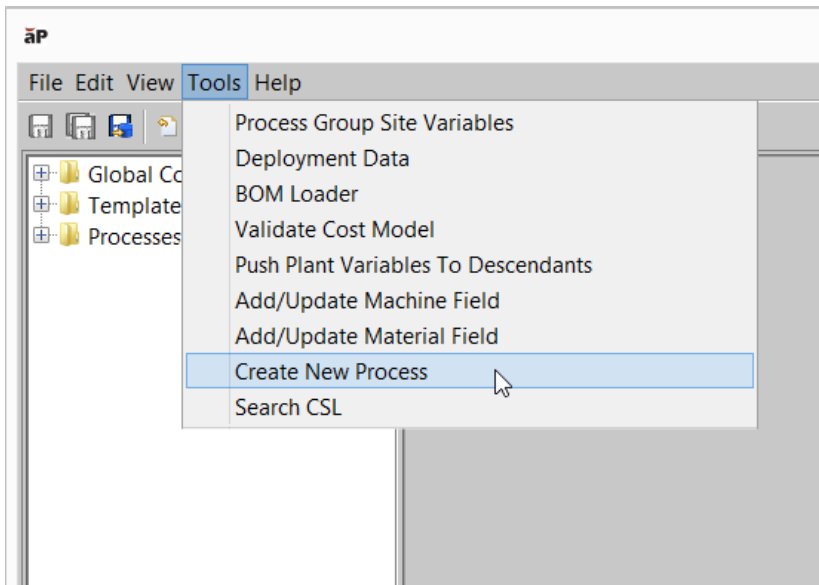
// Account for both types of machine overhead in the system
totalOverheadRate(m) = m.workCenterDirectOverheadRate + m.workCenterIndirectOverheadRate

```

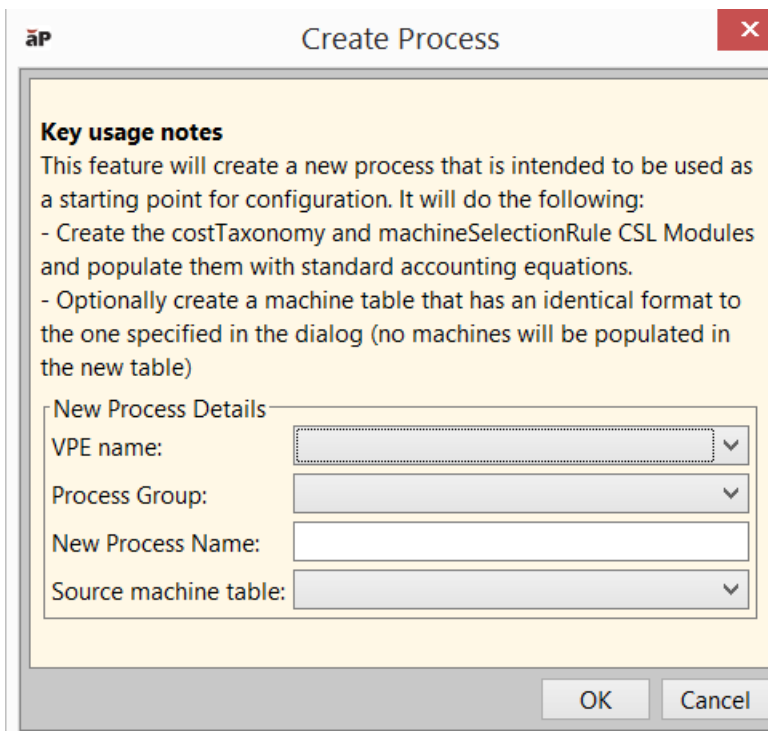
You can also optionally specify a machine type to copy.

Follow these steps:

- 1 Select **Create New Process** from the Tools menu.



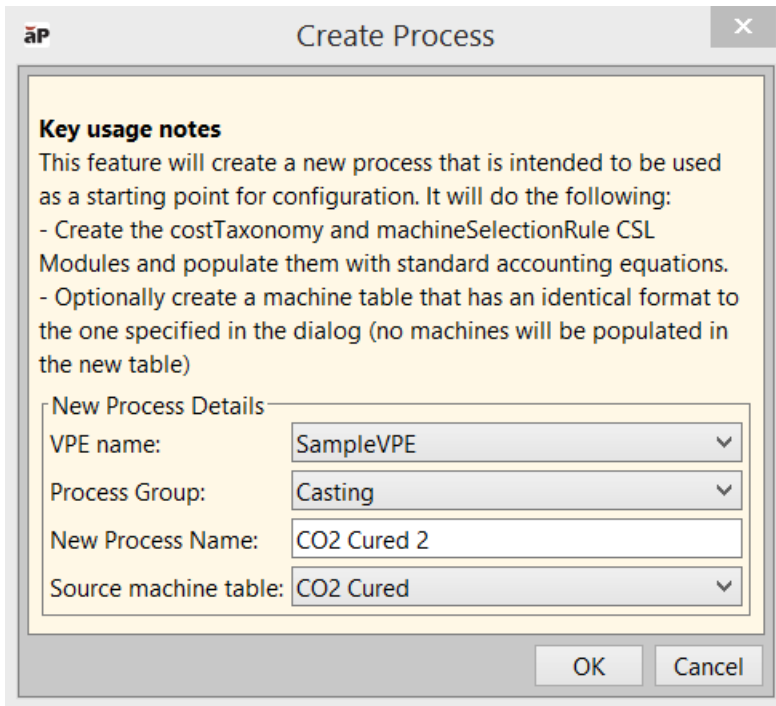
The **Create Process** dialog appears:



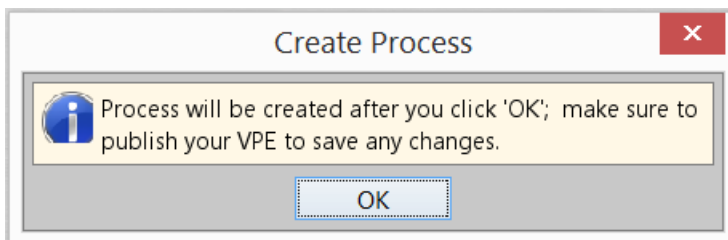
2 Fill in the following dialog fields:


- **VPE name:** VPE to which you want to add a new process. Setting this field populates the dropdown list of choices for the **Process Group** field.
- **Process Group:** process group to which you want to add the new process. Setting this field populates the dropdown list of choices for the **Source machine table** field.
- **New Process Name:** name of the new process.

- **Source machine table:** process whose machine type you want to copy. Only processes in the specified process group are available. Leave this field blank if you don't want to copy a machine type.



- 3 Click **OK** to make the specified changes. A notification dialog appears:

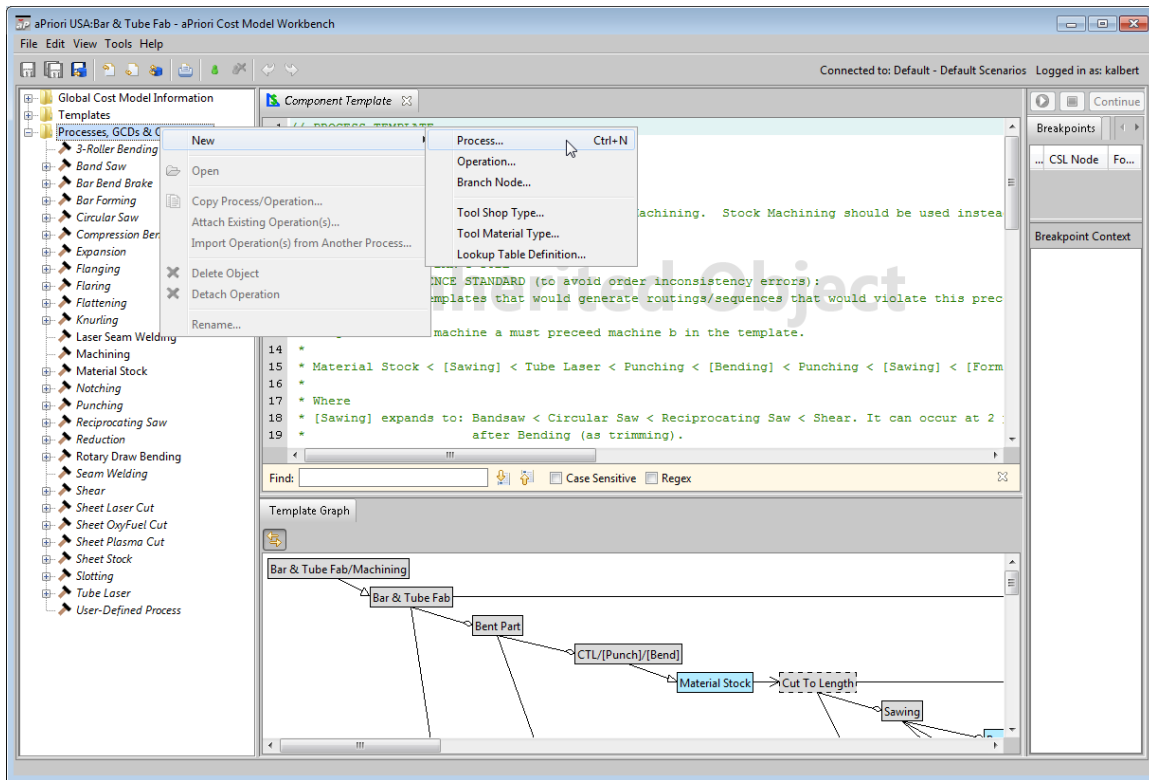


- 4 Click **OK** in the notification dialog. aPriori opens the modified cost model in the CMWB, if it's not already open.
Note: your changes are not saved to your private workspace until you publish the changes to the public cost model (see step 6, below). That is, unless you publish the changes, they will not persist across invocations of aPriori.
- 5 Be sure to modify the template for the component GCD to refer to the new process--see [Working with Templates](#).
- 6 To save the changes to your private workspace and incorporate the changes into the public cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

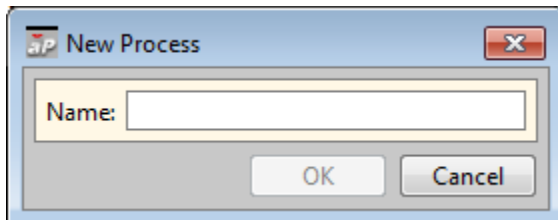
Creating a Process from Scratch

To create a new process from scratch, follow these steps:

- 1 Right-click on any node in the navigation pane, and select **New > Process...** from the context menu. The **New Process** dialog appears.



- 2 Enter a name for the new process.



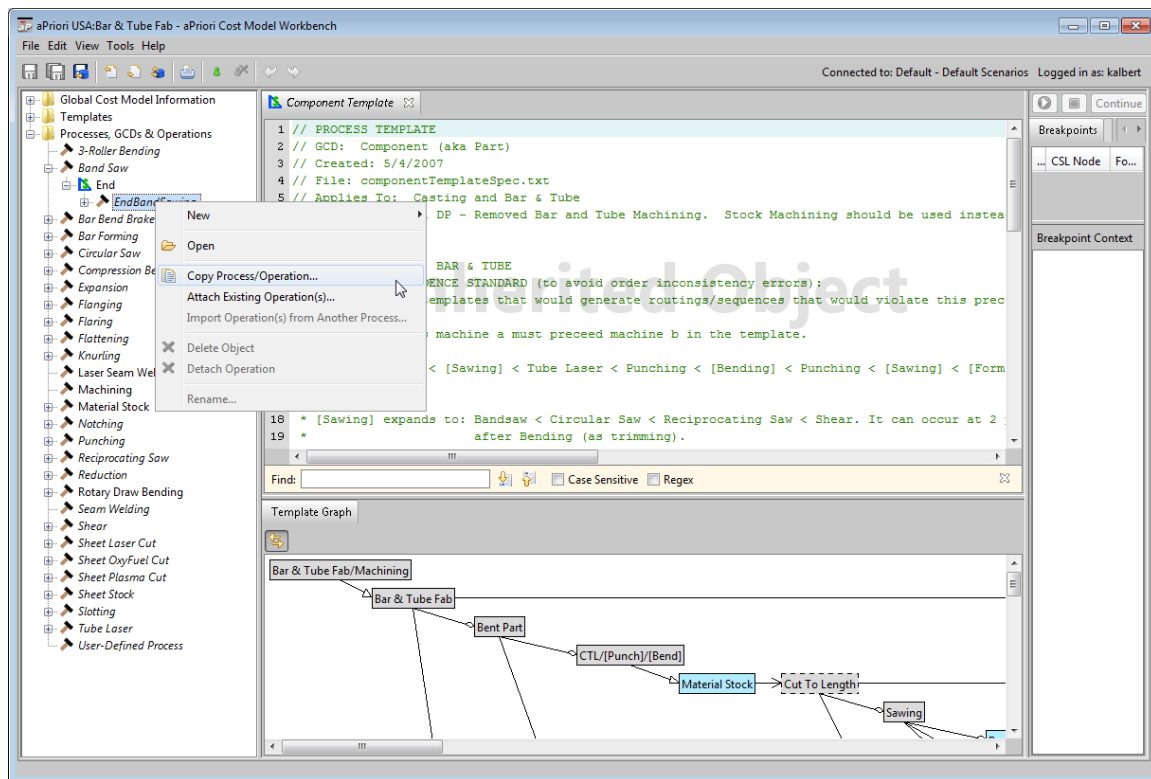
- 3 Modify the template for the component GCD to refer to the new process—see [Working with Templates](#). Create or modify the appropriate CSL modules.

If you want the new process to be a secondary process, define the node attribute **displayinSPDialog** on the node for the new process, and set it to true—see [Working with Node Attributes](#).

Creating and Copying Operations

To create a new operation by copying an existing operation, follow these steps:

- 1 In the navigation pane, under **Processes, GCDs, & Operations**, right-click on the operation you want to copy. Select **Copy Process/Operation...** from the context menu. The **Copy Process** dialog appears.



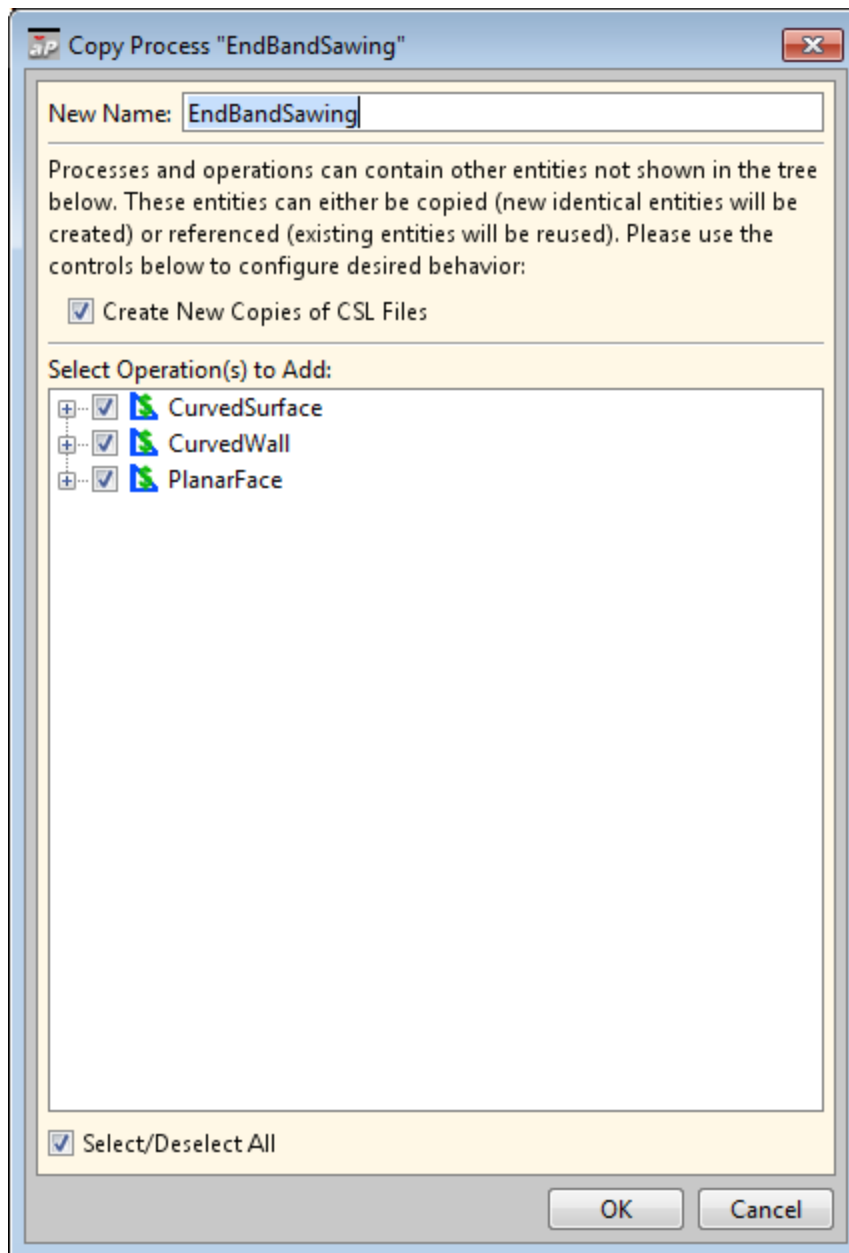
Note that new operation is associated with the same GCD type as the copied operation.

- 2 In the **Copy Process** dialog, modify the following sections, if necessary:

New Name: enter a name for the new operation.

Create New Copies of CSL Files: leave this checked if you want to also copy all the modules associated with some of the operation's potential descendent operations. Uncheck it, if you want to use some of the identical potential descendent operation modules.

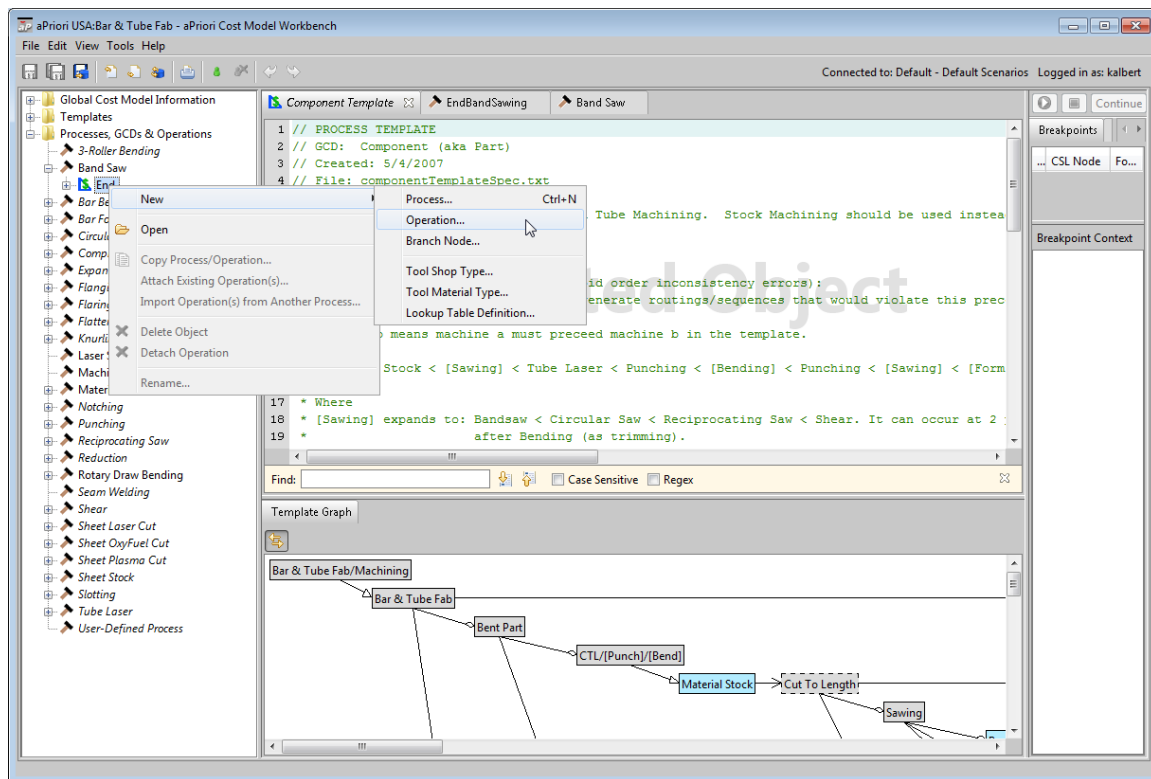
Select Operation(s) to Add: leave checked only those operations that you want to serve as potential descendents of the new operation. Expand the GCD-operation tree, if necessary.



- 3 Modify the template for the new operation's associated GCD to refer to the new operation (and modify the templates for its descendents, if any)—see [Working with .](#) Create or modify the appropriate CSL modules.

To create a new operation from scratch, follow these steps:

- 1 In the navigation pane, under **Processes, GCDs, & Operations**, right-click on the GCD type whose creation the new operation contributes to. Select **New > Operation...** from the context menu. The **New Operation** dialog appears.

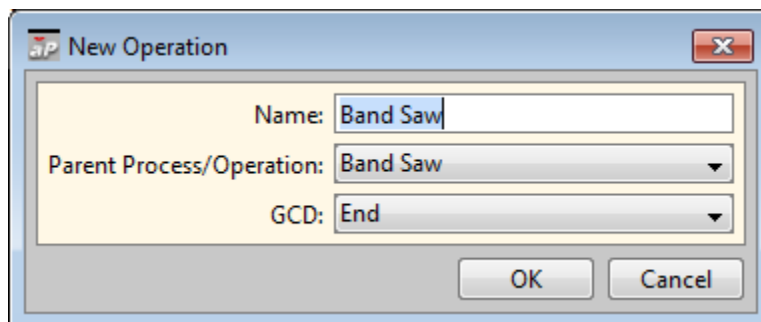


- 2 In the **New Operation** dialog, modify the following sections, if necessary:

Name: enter a name for the new operation.

Parent Process/Operation: specify the parent process or operation.

GCD: specify the new operation's associated GCD type, the GCD type to whose creation the new operation contributes.

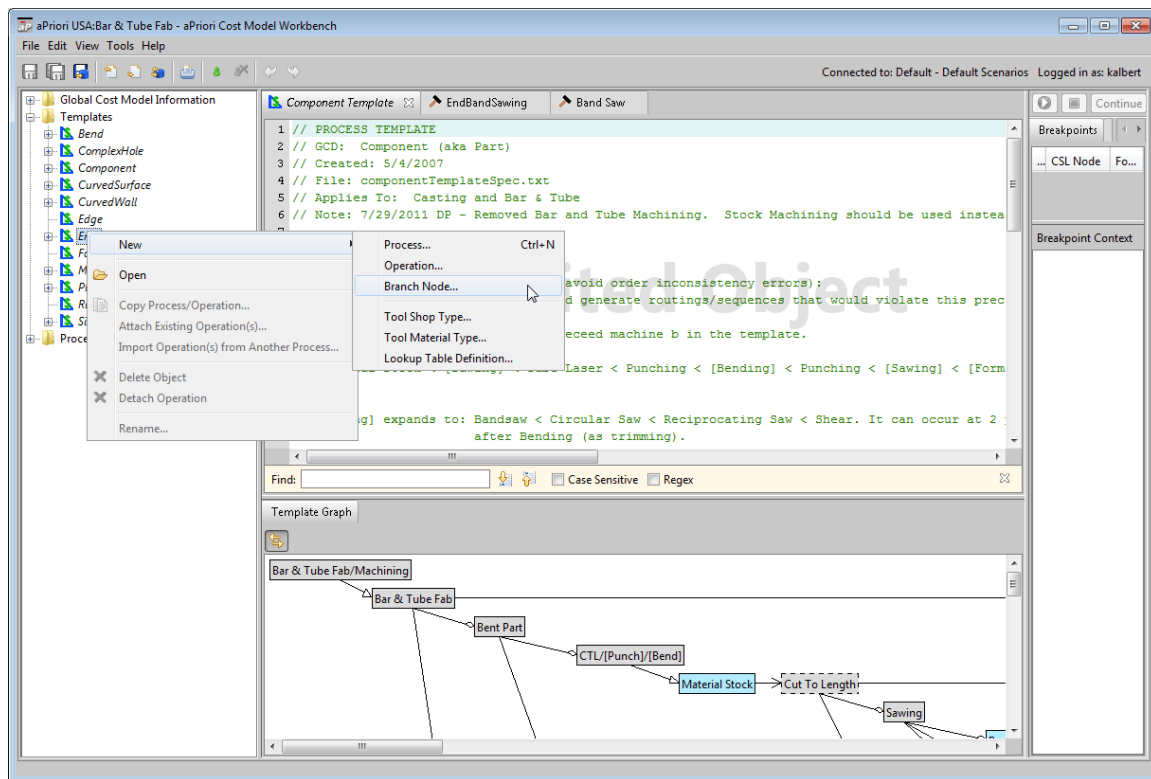


- 3 Modify the template for the new operation's associated GCD to refer to the new operation (and modify the templates for its descendents, if any)—see [Working with](#) . Create or modify the appropriate CSL modules.

Creating Branch Nodes

To create a new branch node, follow these steps:

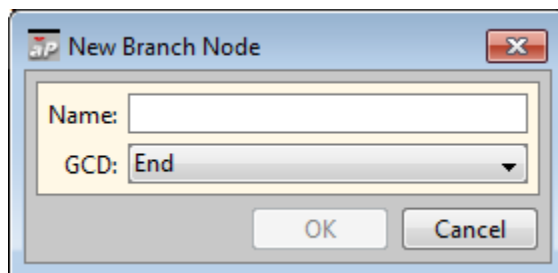
- 1 In the navigation pane, under **Templates**, right-click on the GCD type whose template will refer to the new branch node. Select **New > Branch Node...** from the context menu. The **New Branch Node** dialog appears.



- 2 In the **New Branch Node** dialog, modify the following sections, if necessary:

Name: enter a name for the new operation.

GCD: specify the new branch node's associated GCD type, the GCD type whose template will refer to the new branch node..



- 3 Modify the template for the new branch node's associated GCD type to refer to the new branch node—see [Working with Create or modify the appropriate CSL modules](#).

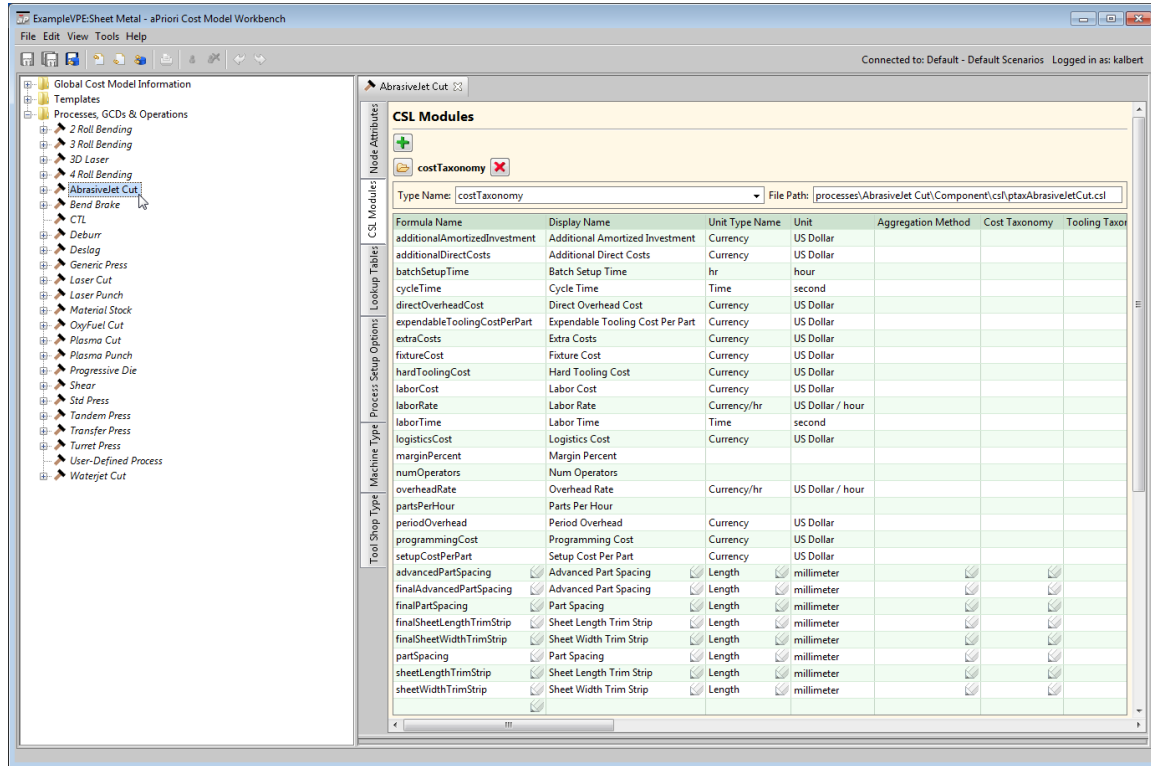
Deleting Processes, Operations, and Branch Nodes



To delete a process, operation, or branch node, follow these steps:

- 1 Delete any references to the process, operation, or branch node, such as CSL modules associated with the process or operation, as well as template references to the process or operation.
- 2 Right-click the node in the CMWB navigation pane, and select **Delete Object** from the context menu.

Working with Formula Tables

Each CSL module has an associated formula table that must list all output formulas (which can be included in results provided by costing tables and reports, as well as in formula dependency trees). If you create or modify a CSL module to include new output formulas, you must modify its associated formula table.




To modify the formula table, you must first select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar; and you must publish your changes when done. Fields with a pencil, , icon are editable.

See the following sections for more information:

- Adding a Formula to the Formula Table
- Controlling Whether a Custom Output Appears in the Part Details Tab

Adding a Formula to the Formula Table

To add a formula, follow these steps:

- 1 Select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 2 Click in the empty Formula Name field of the last line of the table.
- 3 Enter information in the following fields:
 - **Formula Name:** enter name of the formula (one of the names listed above)
 - **Display Name:** enter the name as you want it to appear in end user tables and reports.



- **Unit Type Name:** select the type of units for the formula value.
- **Unit:** units for the formula value. This field is not editable; it is determined by the Unit Type Name field.
- **Inverted Goodness:** set this to false. (This toggles the red/green arrows in the UI. By default ("false"), smaller values are considered desirable (as in the case of costs), and therefore are displayed with green arrows. However, a smaller value for an item such as feed rate could be considered a negative, and setting **Inverted Goodness** to "true" allows it to be displayed with a red arrow.)
- **Display:** set this to true if you want the formula value included in end user tables and reports; set this to false otherwise. See also [Controlling Whether a Custom Output Appears in the Part Details Tab](#).
- **Dependency Tree Visibility:** Set this value to determine the behavior of the right-click **Show Formula Dependencies** command in the UI, so that it displays information that is relevant to the end user. Possible settings are ALWAYS, NEVER, WHEN_NONZERO, or unset. Currently the **Formula Dependencies** window is populated only for direct and indirect rates. The "correct" setting for a given formula is somewhat subjective, but here are some guidelines:
 - WHEN_NONZERO – Use this if formula relevance is determined by another setting such as a cost model variable or a site variable. To have formula relevance be driven by these global toggles, you need to set the visibility to WHEN_NONZERO and (when it is irrelevant in the calculation) force it to evaluate to zero.
 - NEVER – You would typically use this setting only when you have a formula that does not show up in the UI, but which could be referenced in a spreadsheet report. Or for a formula that will be used in PSOs and which is collected by a parent node. Other possible uses would be for a standard formula whose equations are not set up properly to display in the dialog, or for an insignificant calculation such as a "fudge factor".
 - ALWAYS – This is the most typical selection for new formulas. The formula will show up in the dependency tree and the user will be able to see (and override) the value directly from the Formula Dependencies window even if it computes to zero.

Note: If **Dependency Tree Visibility** is unset, the behavior is the same as if it had been set to ALWAYS.

- **Overridable:** set this to true if users should be able to override the value for this formula in the UI.

Note: At the Site Cost Model level, this column is labeled **Overridable At**, and its values define contexts in which the result of the particular formula can be overridden: BRANCH, PROCESS, OPERATION, or UTILIZATION_PROCESS.

- **Description:** enter an optional description of the formula.

- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.



Note that if you've already defined a process setup option that refers to a formula, the formula is automatically added to the formula table.

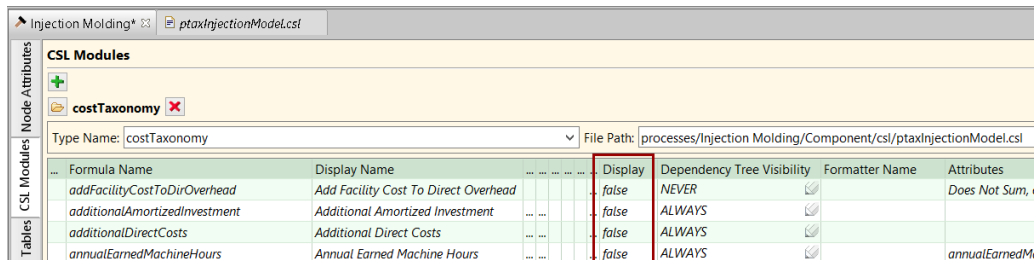
Controlling Whether a Custom Output Appears in the Part Details Tab

In a CSL formula table, a custom output's value for the **Display** column controls whether it appears in the Part Details tab of the aPriori Professional interface.

(Note that to view custom outputs, which are process-group-specific outputs, you must enable a table view that includes **Custom Outputs** in the Displayed Fields column of the Manage Part Details Views dialog—see the section Part Details and Assembly Details and the section Creating New Views in the aPriori Cost Tables chapter of the *aPriori Professional User Guide*.)

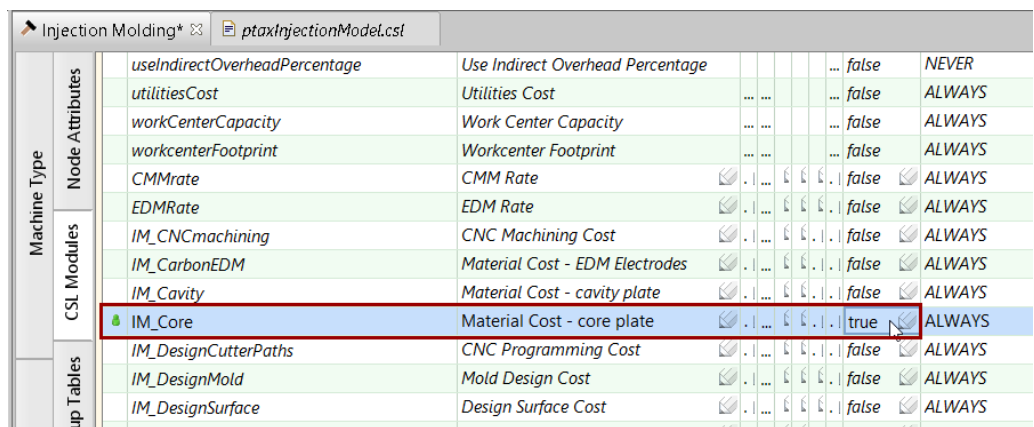
To specify whether a custom output should appear in the Part Details tab, follow these steps:

- 1 Select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 2 Find the output of interest in the formula table, and ensure that it has a pencil, , in the **Display** field. (The pencil indicates that the field is editable.)





Formula Name	Display Name	Display	Dependency Tree Visibility	Formatter Name	Attributes
addFacilityCostToDirOverhead	Add Facility Cost To Direct Overhead	false	NEVER		Does Not Sum, ...
additionalAmortizedInvestment	Additional Amortized Investment	false	ALWAYS		
additionalDirectCosts	Additional Direct Costs	false	ALWAYS		
annualEarnedMachineHours	Annual Earned Machine Hours	false	ALWAYS		annualEarnedMa...

- 3 Set the **Display** field to **true** if you want the formula value included in the Part Details tab; set it to **false** otherwise.



Machine Type	Node Attributes	Formula Name	Display Name	Display	Dependency Tree Visibility	Formatter Name	Attributes
Machine Type	Node Attributes	useIndirectOverheadPercentage	Use Indirect Overhead Percentage	false	NEVER		
		utilitiesCost	Utilities Cost	false	ALWAYS		
		workCenterCapacity	Work Center Capacity	false	ALWAYS		
		workcenterFootprint	Workcenter Footprint	false	ALWAYS		
		CMMrate	CMM Rate	false	ALWAYS		
		EDMRate	EDM Rate	false	ALWAYS		
		IM_CNCmachining	CNC Machining Cost	false	ALWAYS		
		IM_CarbonEDM	Material Cost - EDM Electrodes	false	ALWAYS		
		IM_Cavity	Material Cost - cavity plate	false	ALWAYS		
		IM_Core	Material Cost - core plate	true	ALWAYS		
up Tables	CSL Modules	IM_DesignCutterPaths	CNC Programming Cost	false	ALWAYS		
		IM_DesignMold	Mold Design Cost	false	ALWAYS		
		IM_DesignSurface	Design Surface Cost	false	ALWAYS		

- 4 Select **Save** from the **File** menu, or click , in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click , in the toolbar.

Formulas with **true** in the Display field are included under Custom Outputs in the Part Details tab:

Validation										Part Summary										Design to Cost										Cost Summary										Part Details										Investment									
Cost Object		Cost Des...		Manufacturing Info		Manufact...		Custom Outputs		Compon...		M																																															
Status	Name	Scenario	Locked	Process Group	Labor Time (s)	Output Name	Finish Mass (kg)																																																				
● *	Current	Initial	🔒	Plastic Molding	46		0.44																																																				
●	Plastic Molding				46		0.44																																																				
●	Injection Molding				46		0.44																																																				
						Material Cost - core plate (USD)																																																					
						Cavity Orientation																																																					
						Cooling Time (s)																																																					
						Cycle Time Per Part/Cavity (s)																																																					
						Ejection Time (s)																																																					
						Machine Clamping Force (kN)																																																					
						Mold Size - Height (mm)																																																					
						Mold Size - Length (mm)																																																					

Template Pruning

As with feasibility modules (see [Process and Operation Feasibility](#)), you can use template pruning modules to eliminate routings from consideration by the cost engine. When the cost engine evaluates a feasibility module, it can identify a particular routing as infeasible (and eliminate it from consideration); in contrast, when the cost engine evaluates a pruning module, it can identify a large class of routings as infeasible.

A template pruning module eliminates a node (together with all its descendent nodes) from a routing template, saving the cost engine the trouble of expanding the pruned template portion and preventing consideration of the individual routings that expansion would have produced. For example, a pruning module might eliminate, for parts that have no turning axes, all routings with rotational processes (which are contained in a single subtree of the template).

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has a template pruning module, the module's **Type Name** field is set `templatePruningRule` and the label **templatePruningRule** appears next to the folder icon.

This section includes the following subsections:

- [Context of Evaluation](#)
- [Example](#)

Context of Evaluation

During template expansion, the cost engine visits each template node, and evaluates the node's associated template pruning module, if there is one. If a node's template pruning module returns `false`, the node is removed from the template. All the node's descendents in the template are removed as well. Note that, unlike feasibility modules, template pruning modules are not evaluated in the context of a candidate material stock. See [Cost Engine Details](#).

Example

Consider, for example, the machining process group in aPiori starting point VPEs. The process routing template node `Rotational` has an associated pruning module. This module eliminates all turning routings for parts that have no valid turning axis.

Here is a simplified version of this file:

```
Rule HasTurningAxes: (countTurningAxes > 0)
Message HasTurningAxes: 'PRUNE: ' + _
    'Skip Turning Routings: this part has no turning axes and would
    not benefit.'

// Total count of turning axes extracted
countTurningAxes = _
    select count(x) from part.childArtifacts x where isTurningAxis(x)
```

This module has one rule and one function definition.

The function uses a query to count the current part's turning axes. The query effectively forms the set of child GCDs that are turning axes, and then returns the number of elements in that set. The query is composed of these expressions:

- `count(x)`: Query aggregate function (see Query Expressions in CSL chapter) that returns the number of elements in the collection formed by the query.
- `part.childArtifacts`: Object access that uses the `childArtifacts` attribute of the current part (designated by the standard CSL input `part`) to retrieve the collection of the current part's child GCDs.
- `isTurningAxis(x)`: builtin CSL function that returns true if the specified GCD is a turning axis; returns false otherwise. This filters the child GCDs, so that only those that are turning axes are included in the collection formed by the query.

The rule tests whether the function result (the number of turning axes that are child GCDs of the current part) is greater than 0. If the rule evaluates to true, the `Rotational` node is retained in the template, and the cost engine generates the routing that it (and its descendents) represent. If the rule evaluates to false, the `Rotation` node (together with all its descendents) is pruned, and the cost engine never generates the individual routings that it represents.

Material Stock Selection

Some process groups use material stock. The material that is used by such process groups has some form or structure even before the manufacturing process begins. For the sheet metal process group, for example, the material takes the form of a metal sheet; for plastic molding, in contrast, the material takes the form of an unstructured volume or mass of plastic pellets.

For process groups that use material stock, the cost model data specifies a list of available stocks for each type of material (see Working with Cost Model Data and

Metadata). For a particular part, the end user can explicitly specify the stock to use; alternatively, the end user can let aPriori select the stock. For cases in which the end user has elected to let aPriori select the material stock, you can supply a material stock selection module to narrow down the available stocks to those that are feasible or desirable for the current part.

For example, for sheet metal parts, a stock selection module can check the geometry of the part, and eliminate stocks whose dimensions are exceeded by the blank dimensions.

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has a material stock selection module, the module's **Type Name** field is set `materialStockSelector` and the label **materialStockSelector** appears next to the folder icon.

This section includes the following subsections:

- [Context of Evaluation](#)
- [Finding or Designating a Routing's Stock Selector](#)
- [Stock Selector Input](#)
- [Stock Selector Outputs](#)
- [Example](#)

Context of Evaluation

After the cost engine expands process templates, it considers each resulting routing in turn. For each routing, it evaluates the routing's stock selector, and then for each resulting stock, it visits each process in the routing in order to perform process feasibility checks, machine selection, operation assignment, and operation and process costing (see [Cost Engine Details](#)).

So each stock selector is evaluated in the context of a particular process routing. But, for that routing, machines have not yet been selected, operations have not been assigned, and no processes or operations have been costed.

Finding or Designating a Routing's Stock Selector

A process routing's stock selection module is associated with one of the routing's nodes. Typically, a routing's stock selector is associated with the process called **Material Stock**. In some cases, to allow different routings to have different stock selectors, multiple nodes have stock selector modules, but only one is evaluated by the cost engine for a given routing.

The simplest way for a routing to designate the stock selector node is by the presence of the node attribute **utilizationProcess**, where the attribute has no value. If there is more than one node with this attribute (and they are all valueless), the cost engine treats the right-most active one as the stock selector node.

A routing can also designate the stock selector node by specifying the node's name as the value of **utilizationProcess** for some other node in the routing. If there is more than one node with a value for this attribute, the left-most one specifies the name of the stock selector node.

For example, for the sheet metal process group, if a routing includes the branch node `Prog Die`, which has a node attribute **utilizationProcess** whose value is `Progressive`

Die, the cost engine evaluates the stock selection module associated with `Progressive Die` (assuming no other node has a value for `utilizationProcess`).

See also Cost Engine Details.

Stock Selector Input

For process groups that use material stock, the complete list of available stocks (for the current user-selected material) is presented in aPiori's **Material Selection** dialog, when the **Material Stock** pane is expanded. When the cost engine evaluates a stock selection module, it puts into scope the CSL standard input `stocks`, and establishes its value as a collection that contains each of these available stocks.

Stock Selector Outputs

If the stock selection module establishes a collection of stocks as the value for the CSL output `validMaterialStocks`, the cost engine considers each stock in the collection.

If the stock selection module does not establish a non-null value for the CSL output `validMaterialStocks`, the cost engine considers only one, virtual stock, specified as follows:

- For the Machining process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:
 - Length:
 - CSL output `virtualStockLength`, if non-null.
 - Otherwise, the larger of the current part's `minStockLength` and the value of the plant variable `standardStockLength`, if the plant variable is defined.
 - Otherwise, the larger of the current part's `minStockLength` and 20 feet.
 - Width:
 - CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the width of the current part's cross-section.
 - Height:
 - CSL output `virtualStockHeight`, if non-null.
 - Otherwise, the height of the current part's cross-section.
 - Inside diameter:
 - CSL output `virtualStockInsideDiameter`, if non-null.
 - Otherwise, the inside diameter of the current part's cross-section.
 - Outside diameter:
 - CSL output `virtualStockOutsideDiameter`, if non-null.
 - Otherwise, the outside diameter of the current part's cross-section.
 - Thickness:
 - CSL output `virtualStockThickness`, if non-null.
 - Otherwise, the thickness of the current part's cross-section..
- For the Sheet Metal process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:

- Length:
 - The larger of the blank's length and the CSL output `virtualStockLength`, if non-null.
 - Otherwise, the larger of the blank's length and the value of the plant variable **standardStockLength**, if the plant variable is defined.
 - Otherwise, the larger of the blank's length and 8 feet.
- Width:
 - The larger of the blank's width and the CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the larger of the blank's width and the value of the plant variable **standardStockWidth**, if the plant variable is defined.
 - Otherwise, the larger of the blank's width and 4 feet.

Example

In aPriori's starting point VPEs, the sheet metal process Material Stock uses a stock selection module to find stocks with a length, width, and thickness appropriate for the current part:

```
THICKNESS_SNAP = 0.05 // mm error between the actual measured sheet
metal thickness and the closest actual stock size

validMaterialStocks = select from stocks as s where _
  equalsEps(s.thickness, part.thickness, THICKNESS_SNAP) and _
  ((part.blankBoxLength <= s.length and part.blankBoxWidth <=
s.width) or _
  ((part.blankBoxLength <= s.width and part.blankBoxWidth <=
s.length)))
```

This query uses the following simple identifiers:

- `validMaterialStocks` is the module's output.
- `stocks` is the CSL standard input containing each initially available stock.
- `s` is, in effect, bound to each element of `stocks` in turn.
- `part` is the CSL standard input designating the current part.
- `THICKNESS_SNAP` is a constant, the maximum allowable difference between stock and part thickness.

The query assigns to `validMaterialStocks` a collection that contains the elements of stock that meet both the following conditions:

- The stock's thickness is within `THICKNESS_SNAP` of the part's thickness.
- The part's blank fits on the stock, that is one of the following is true:
 - The blank's length and width are less than the stock's length and width

- The blank's length is less than the stock's width and the blank's width is less than the stock's length.

Process and Operation Optionality

For optional processes and operations, optionality modules can help determine whether the cost engine considers the process or operation to be *required* in order for the current process routing or operation sequence to make the current part or GCD. Here is how being required affects routing:

- If a process or operation is not required, it is removed from the current routing or sequence.
- If a process or operation is required and feasible, it is included in the current routing or sequence.
- If a process or operation is required and not feasible, the current routing or sequence fails.

A process or operation is required if and only if it is not user-excluded (see the aPriori User Guide) and any of the following is true of it:

- Is user included (see the aPriori User Guide).
- Has a taxonomy module whose evaluation yields no results.
- Has an optionality module that returns `true`.
- Has at least one required (in the sense described here) child in the process-operation hierarchy (see Cost Engine Details).

The code in an optionality module should specify the conditions under which the associated process or operation is required to be included in a process routing or operation sequence in order for that routing or sequence to successfully create a given part or GCD (assuming there are no required children of the process or operation—if there are required children, the process or operation is required, regardless of what its optionality module returns).

Only an optional node can have an optionality module.

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has an optionality module, the module's **Type Name** field is set `optionalProcess` or `optionalOperation`, and the label **optionalProcess** or **optionalOperation** appears next to the folder icon.

Optionality module names start with `opt` and end with `.csl`.

This section includes the following subsections:

- [Context of Evaluation](#)
- [Example](#)

Context of Evaluation

These types of modules are evaluated in the context of a process routing or operation sequence that includes the current node.

The cost engine has established a process-operation hierarchy that includes the nodes in the current routing or sequence and all their ancestors. The cost engine has not yet

established the portion of the process-operation hierarchy below the nodes in the current routing or sequence. The cost engine might not have yet established the portion that includes operations to create a given sibling (in the GCD hierarchy) of the current GCD.

The cost engine has chosen a particular candidate material stock (if relevant). A machine or tool for the current node has not yet been chosen, and no nodes have yet been costed.

Example

Here is a machining optionality module for the operation branch node `Edge Treatments` in the template for the GCD type `Edge`. This node stands for operations such as deburring, rounding, and chamfering. This module checks to see if any such operations are required for the current GCD.

Plant variables control whether aPiori should automatically determine whether deburring is necessary. The cost model assumes that deburring is necessary for sharp edges that were *not* created either by turning or (for hole edges) by a sheet metal operation.

Other kinds of edge treatment for hole edges are assumed to be necessary for round edges that are part of non-flanged holes and for chamfers that are part of holes that have not been previously countersunk.

The code divides edge treatment operations into the following categories:

- Non-hole edge deburring: This is required if and only if all the following are true of the edge:
 - External edge and the plant variable `autoSelectExternalEdgesForDebur` is set to `true` or is an internal edge and plant variable `autoSelectInternalEdgesForDebur` is set to `true`.
 - Sharp.
 - Not a child GCD of a simple hole.
 - Has no coincident turning axis (and so is not created by turning operation).
- Hole edge deburring: This is required if and only if all the following are true of the edge:
 - Sharp.
 - Not created by a sheet metal operation.
 - Child GCD of a simple hole.
 - Plant variable `autoDeburSharpHoleEdges` is set to `true`.
 - Has no coincident turning axis (and so is not created by turning operation).
- Hole edge machining: This is required if and only if either of the following is true of the edge:
 - Round and non-flanged.
 - Chamfer and not previously countersunk.

```
import libSetups.csl
```

```
Rule IsEdgeTreatmentsRequired: _
    IsEdgeDeburringRequired or IsHoleEdgeDeburringRequired or _
```

```

    IsHoleEdgeMachiningRequired
Message IsEdgeTreatmentsRequired: OLT + _
    'Edge treatments, e.g., rounding, chamfering, countersinking, ' _
    'edge turning or deburring, are not required'

IsEdgeDeburringRequired = _
    {true if ( _
        ( _
            and _                (plant.autoSelectExternalEdgesForDebur == true
                                gcd.internal == false)_
            or _
            (plant.autoSelectInternalEdgesForDebur == true
            and _                gcd.internal == true) _
        )
        and _
        gcd.edgeTypeName == 'SHARP' _
        and _
        hasNoParentGcd _
        and _
        GetCoincidentTurningAxis(gcd) == null
    ) _
    false otherwise}

IsHoleEdgeDeburringRequired = _
    {true if _
        gcd.edgeTypeName == 'SHARP' and _
        part.processGroupName != 'Sheet Metal' and _
        hasParentGcd and plant.autoDeburrSharpHoleEdges ==true and _
        GetCoincidentTurningAxis(gcd.parentArtifact) == null _
    false otherwise}

IsHoleEdgeMachiningRequired = _
    {true if ( _
        (gcd.edgeTypeName == 'ROUND' and not(isFlangedHoleEdge))) or _
        (gcd.edgeTypeName == 'CHAMFER' and
        not(previousCounterSunkEdge)
    ) _

```

```

false otherwise}

previousCounterSunkEdge =
  {hasNodeInTree (op.parentArtifactResult, 'PreviouslyCountersunk')
  -
  if ( _
    op.parentArtifactResult != null and _
    op.parentArtifactResult.artifactTypeName == 'SimpleHole' and _
    gcd.parentArtifact.isCountersunk == true _
  ) _
  false otherwise _
}

isFlangedHoleEdge = _
  {true if _
    gcd.parentArtifact != null and _
    gcd.parentArtifact.artifactTypeName == 'SimpleHole' and _
    gcd.parentArtifact.isFlanged == true
  false otherwise _
}

hasNoParentGcd = _
  {false if gcd.parentArtifact.artifactTypeName == 'SimpleHole'
  true otherwise}

hasParentGcd = _
  {true if gcd.parentArtifact.artifactTypeName == 'SimpleHole'
  false otherwise}

```

The rule in this module refers to three boolean-valued formulas, corresponding to the three categories described above:

- IsEdgeDeburringRequired
- IsHoleEdgeDeburringRequired
- IsHoleEdgeMachiningRequired

IsEdgeDeburringRequired refers to the formula `hasNoParentGCD`. The formula, defined later in the module, is true if and only if the current GCD is not part of a simple hole.

IsHoleEdgeDeburringRequired refers to the formula `hasParentGCD`. The formula, defined later in the module, is true if and only if the current GCD is part of a simple hole.

IsHoleEdgeMachiningRequired refers to the following formulas:

- `previousCountersunkEdge`: true if the current edge is part of a previously countersunk simple hole; false otherwise.
- `isFlangedHoleEdge`: true if the current edge is a part of a flanged, simple hole; false otherwise.

The formulas use the following CSL standard inputs:

- `gcd`: current GCD. has one field for each attribute of the current GCD. This code uses the following attributes:
 - `artifactTypeName`: names the specified GCD's type; string valued. Used here to determine if the edge's parent is a simple hole.
 - `edgeTypeName`: is used to determine if the specified GCD is a sharp edge, round, or chamfer; string valued.
 - `internal`: indicates whether the specified GCD is an internal or external edge; boolean valued.
 - `isFlanged`: true if the specified GCD is flanged; false otherwise.
 - `isCountersunk`: true if the specified GCD is countersunk; false otherwise.
 - `parentArtifact`: parent of the specified GCD in the GCD hierarchy.
- `op`: Current operation. fields include the following:
 - `parentArtifactResult`: parent of the current node in the process-operation hierarchy.
- `part`: Current part. fields include the following:
 - `processGroupName`: names the current *primary* process group; string valued.
- `plant`: has one field for each plant variable. This code accesses the fields for the following plant variables:
 - `autoDeburSharpHoleEdges`
 - `autoSelectExternalEdgesForDebur`
 - `autoSelectInternalEdgesForDebur`

The formulas also use the following functions:

- `GetCoincidentTurningAxis`, defined in the imported library file `libSetups.csl`. The function returns `null` if and only if there is no turning axis that is coincident with the argument's axis.
- `hasNodeInTree`: CSL predefined function. Returns true if the specified node or one of its descendents has a node attribute named by the specified string.

Process and Operation Feasibility

For a non-optional process or operation, feasibility modules are used to specify two kinds of conditions:

- Conditions under which it is *possible* (in the context of the current routing) to use the process or operation in the creation of the current part. If the conditions aren't met, the current routing fails.
- Conditions under which it is *required* (in the context of the current routing) to use the process or operation in the creation of the current part. If the conditions aren't met, the current routing fails.

For an optional process or operation, the feasibility module specifies just the first kind of condition (conditions under which it is possible); the optionality module specifies the second kind (see [Process and Operation Optionality](#)).

The module should return true if the conditions are met, and return false otherwise.

So for example an injection molding feasibility module might check for compatibility between the process and the current material; the feasibility module for a tube laser hole cutting operation might check that the hole is large enough and not flanged, threaded, or countersunk (see the examples below).

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has a feasibility module, the module's **Type Name** field is set `routingRule`, and the label **routingRule** appears next to the folder icon.

Feasibility module names should start with `pfr` (for processes) or `ofr` (for operations) and end with `.csl`.

This section includes the following subsections:

- [Context of Evaluation](#)
- [Examples](#)

Context of Evaluation

Feasibility modules are evaluated in the context of a process routing or operation sequence that includes the current node.

The cost engine has established a process-operation hierarchy that includes the nodes in the current routing or sequence and all their ancestors. The cost engine has not yet established the descendents in the process-operation hierarchy of the nodes in the current routing or sequence. The cost engine might not have yet established the portion of the hierarchy that includes operations to create a given sibling of the current GCD.

The cost engine has chosen a particular candidate material stock (if relevant). A machine or tool for the current node has not yet been chosen, and no nodes have yet been costed.

Examples

In aPiori's starting point cost model for plastic molding, the feasibility module for the Injection Molding process checks that the current material is compatible with injection molding.

```
Rule CompatibleMaterial: material.canIM_SFM
Message CompatibleMaterial: _
    'Failed because you cannot Injection Mold this type of material'
```

This rule uses the CSL standard input `material`, which has one field for each attribute of the current material (as listed in the **Material Composition** table of aPiori's **Material Selection** dialog). The attribute `canIM_SFM` is set to `true` for thermoplastics and `false` for thermosetting materials.

In the Bar&Tube process group, the operation that uses a tube laser to create a simple hole has the following feasibility module:

```
import libGtolProcessCapabilities.csl

Rule GtolValueOk: IsProcessAllGtolCapable(gcd, 'Laser Cut')
Message GtolValueOk: IsProcessAllGtolCapableMsg(gcd, 'Laser Cut')

Rule NonPositiveDiameter: gcd.diameter > 0
Message NonPositiveDiameter: _
    'Cant laser cut a simple hole with a non-positive diameter: ' + _
        diameter + ' mm'

diameter = roundEps(gcd.diameter, 0.01)

Rule NonPositiveLength: gcd.length > 0
Message NonPositiveLength: _
    'Can t laser cut a simple hole with a non-positive length
    (depth): ' + _
        holeLength + ' mm'

holeLength = roundEps(gcd.length, 0.01)

Rule IsFlanged: (not gcd.isFlanged)
Message IsFlanged: _
    'Hole is flanged so cannot be completed by laser cutting'

Rule isUnthreaded: gcd.threaded == false
Message isUnthreaded: _
    'The hole is threaded. Laser cutting alone will not make a
    threaded hole'

Rule isNotCountersunk: gcd.isCountersunk == false
Message isNotCountersunk: _
    'The hole is countersunk. Laser Cutting alone will not make a
    countersunk hole'

Rule isNotBlind: gcd.isBlind == false
Message isNotBlind: _
    'The hole is blind. A blind hole cannot be laser cut'
```

The rules use the following fields of the CSL standard input `gcd`:

- `diameter`
- `length`
- `tolerance`
- `isFlanged`
- `threaded`
- `isCountersunk`
- `isBlind`

The messages rely on formulas that round off the numerical values to the nearest 1/100th.

The function `IsProcessAllGtolCapable` is defined in the library `libGtolProcessCapabilities.csl`.

Machine Selection

A machine selection module helps aPriori choose a machine that is feasible and desirable for a given process. A plastic molding machine selection module might, for example, find the machine with lowest overhead rate that is large enough to accommodate the current part and has sufficient clamp force and shot size to make the part (see the example below).

The job of the module is to select a machine and assign it to the CSL output `machine`.

The list of available machines for a given process is presented in aPriori's **Edit Machine Selection** dialog (right click on a process in the manufacturing process pane). When the cost engine evaluates a machine selection module, it puts into scope the CSL standard input `machines`, and establishes as its value either a collection that contains each of these available machines or a collection that contains just the user-selected machine, depending on the machine selection mode.

aPriori has four machine selection modes (see the **Edit Machine Selection** dialog in aPriori):

- **aP Select**
- **User Select/ if not feasible, fail to cost**
- **User Select/ if not feasible, auto-select**
- **User Select/do not check feasibility**

In **aP Select** mode, the cost engine establishes the value of `machines` as a collection all available machines. If the module fails to establish a non-null value for the CSL output `machine`, the current process routing is deemed infeasible.

In **User Select/ if not feasible, fail to cost** mode, the machine selection module essentially serves to evaluate the feasibility of the user-selected machine. In this mode, the cost engine establishes the value of `machines` as a collection containing just the user-selected machine. If the user-selected machine is feasible, the module assigns it the CSL output `machine`. If the user-selected machine is not feasible, the module leaves the value of `machine` null. In this case, the current process routing is deemed infeasible.

In **User Select/ if not feasible, auto-select** mode, the machine selection module potentially serves two purposes: to evaluate the feasibility of the user-selected machine, and (if the module determines that the user-selected machine is infeasible), to select a machine from the list of all available machine. In this mode, the cost engine might evaluate the module twice. The cost engine first assigns to `machines` a collection containing only the user-selected machine. If the module fails to establish a non-null value for the CSL output `machine`, the cost engine evaluates the module again, this time assigning to `machines` a collection containing the full list of available machines. If the module still fails to establish a non-null value for `machine`, the current process routing is deemed infeasible.

In **User Select/do not check feasibility** mode, the cost engine does not evaluate the machine selection module.

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has a machine selection module, the module's **Type Name** field is set to `machineSelectionRule` and the label `machineSelectionRule` appears next to the folder icon.

Machine selection module names should start with `mse1` and end with `.csl`.

This section includes the following subsections:

- [Context of Evaluation](#)
- [Example](#)

Context of Evaluation

These types of modules are normally evaluated in the context of a process routing (and possibly a material stock) and a particular routing node, but operation sequences have not yet been chosen, and no process has yet been costed.

For node's that have a **selectMachineAfterOpAssignment** node attribute set to a non-null value, machine selection is delayed, and performed just before evaluation of the node's taxonomy module.

Before performing machine selection for a node, the cost engine checks the node's ancestors in the current routing to see if any has a machine selection module of its own. If one does, the ancestor machine applies to the current node, and the cost engine does *not* evaluate the current node's machine selection module (if it has one).

Example

Below is a portion of the injection molding machine selection module for aPiori's plastic molding starting point cost model. The module finds the machine with the lowest overhead rate that satisfies these criteria:

- Large enough for the part's required mold.
- Provides sufficient clamp force for the part and the number of mold cavities.
- Can provide the required shot size.

The module assigns to the CSL output `machine` the result of applying the function `selectMachine` to the mold base dimensions.


```
machine = selectMachine (moldBaseXmm, moldBaseYmm)
```

```
selectMachine (moldBaseXmm, moldBaseYmm) = _
  select first(m) from machines m _
  where _
    ClampForceCheck and _
    tieBarHCheck and _
    tieBarVCheck and _
    moldHeightCheck and _
    shotSizeCheck _
  order by m.workCenterOverheadRate
```

The function `selectMachine` performs a query that does the following:

- Selects the machines that satisfy the various checks (mold size, clamp force, and shot size), and
- Orders the machines by overhead rate (as specified by `order by m.workCenterOverheadRate` in the `order by` clause), from lowest to highest.

The final query result is the first of the selected machines in the specified order (as specified by the `first(m)` in the `select` specification), since it is the least expensive one.

The query contains the following simple identifiers:

- `machines`: CSL standard input
- `m`: in effect, bound to each element of `machines` in turn. This variable remains in scope in the rules referred to by the query. The module uses the following fields of `m`, which are all machine attributes (found in aPriori's **Edit Machine Selection** dialog):
 - `workCenterOverheadRate`
 - `tieBarDistanceH`
 - `name`
 - `tieBarDistanceV`
 - `maxMoldHeight`
 - `shotSize`
 - `clampForce`
- `ClampForceCheck`: rule (shown below) in the same module that compares the machine clamp force with the required clamp force. Required clamp force is calculated by a formula in the same module, shown below.
- `tieBarHCheck`: rule (shown below) in the same module that compares the machine and mold dimensions, shown below.
- `tieBarVCheck`: rule (shown below) in the same module that compares the machine and mold dimensions, shown below.
- `moldHeightCheck`: rule (shown below) in the same module that compares the machine and mold dimensions, shown below.

- `shotSizeCheck`: rule (shown below) in the same module that compares the maximum machine shot size with required shot size. Required shot size is calculated by a formula in the same module, shown below.

Some values are rounded off to less precise values for use in messages.

```
Rule tieBarHCheck: m.tieBarDistanceH >= moldBaseXmm
```

```
Message tieBarHCheck: m.name + _
  ' is not feasible. The mold base horizontal dimension ' + '(' + _
  roundMoldBaseX + _
  'mm ) is greater than the horizontal distance between the tie
  bars (' + _
  maxTieBarDistanceH + 'mm)'
```

```
roundMoldBaseX = roundEps(moldBaseXmm, 1)
```

```
maxTieBarDistanceH = m.tieBarDistanceH
```

```
Rule tieBarVCheck: m.tieBarDistanceV >= moldBaseYmm
```

```
Message tieBarVCheck: m.name + _
  ' is not feasible. The mold base vertical dimension ' + '(' + _
  roundMoldBaseY + _
  'mm ) is greater than the vertical distance between the tie bars
  (' + _
  maxTieBarDistanceV + 'mm)'
```

```
roundMoldBaseY = roundEps(moldBaseYmm, 1)
```

```
maxTieBarDistanceV = m.tieBarDistanceV
```

```
Rule moldHeightCheck: m.maxMoldHeight >= moldBaseHeight
```

```
Message moldHeightCheck: m.name + _
  ' is not feasible. The required mold base height ' + '(' + _
  roundMoldBaseHeight + _
  'mm ) is greater than the maximum mold height for this machine ('
  + _
  maxMoldHeight + 'mm)'
```

```
roundMoldBaseHeight = roundEps(moldBaseHeight, 1)
```

```
maxMoldHeight = m.maxMoldHeight
```

```

moldBaseHeight = 2 * part.boxHeight

Rule shotSizeCheck: m.shotSize >= requiredShotSize

Message shotSizeCheck: m.name + _
    ' is not feasible. The required shot size ' + _ + '(' + _
    roundrequiredShotSize + _
    'g ) is greater than the maximum shot size for this machine (' + _
    _
    maxShotSize + 'g of polystyrene) '

roundrequiredShotSize = roundEps(requiredShotSize , 1)
maxShotSize = m.shotSize //units are in grams of polystyrene
requiredShotSize = plant.shotSizeSafetyFactor * _
    (part.volume * numCavities *
    plant.densityPolystyreneForShotSize)/1000000

Rule ClampForceCheck: m.clampForce >= requiredForce

Message ClampForceCheck: m.name + _
    ' is not feasible. The required clamp force ' + _
    '(' + roundRequiredForce + _
    'kN ) is greater than the max clamp force of this machine (' + _
    maxClampForce + 'kN) '

roundRequiredForce = roundEps(requiredForce, 1)
maxClampForce = m.clampForce

requiredForce = _
    ((part.projectedArea * numCavities) + runnerArea) * _
    material.clampingMNPerSqM * plant.clampForceSafetyFactor /
    1000

runnerArea = numCavities * (partLength + partWidth) *
    nominalWallThickness

```

Portions of the module not shown above retrieve or calculate the following:

- moldBaseXmm, moldBaseYmm: horizontal and vertical mold base dimensions.
- numCavities: number of mold cavities (user-specified or based on required number to satisfy production requirements)

- `nominalWallThickness`: Number to summarize part wall thicknesses for cool time and clamp force calculations.

Tool Selection

A tool selection module helps aPiori choose a tool that is feasible and desirable for a given operation. A machining tool selection module might, for example, find the lowest cost tool with an adequate reach, hardness, diameter, and cutting speed (see the example below).

The job of the module is to select a tool and assign it to the CSL output `tool`.

When the cost engine evaluates a tool selection module, it puts into scope the CSL standard input `tools`, and establishes as its value a collection that contains each of the available tools.

When you navigate to the CSL modules for a node (see [Navigating from the Template Graph to the Data for a Given Node](#)), if the node has a tool selection module, the module's **Type Name** field is set to `toolLookup` and the label **toolLookup** appears next to the folder icon.

Tool selection module names should start with `tool` and end with `.csl`.

Tool selection modules are evaluated in the context of a process routing or operation sequence that includes the current node.

The cost engine has established a process-operation hierarchy that includes the nodes in the current routing or sequence and all their ancestors. The cost engine has not yet established the descendents in the process-operation hierarchy of the nodes in the current routing or sequence. The cost engine might not have yet established the portion of the hierarchy that includes operations to create a given sibling of the current GCD.

The cost engine has chosen a particular candidate material stock (if relevant). No nodes have yet been costed.

Process and Operation Taxonomy

Taxonomy modules contain the heart of a cost model's logic. They calculate the costs in various categories (such as tooling costs, labor costs, and overhead costs), associated with performing a process or operation. In order to calculate costs, these modules calculate times (such as cycle times), material utilization, and other physical and monetary quantities. The outputs of a taxonomy module are specified in the module's associated formula table (see [Navigating from the Template Graph to the Data for a Given Node](#)), and are presented in aPiori's **Part Details** tab.

The Site Cost Model taxonomy file (see [Root and Site Cost Model Formulas](#) in the [Cost Model Guide](#)) aggregates the costs across processes, for presentation in aPiori's **Cost Summary**, **Part Details**, **Investment** tabs. Each process or operation aggregates the costs and other physical quantities across its children in the process-operation hierarchy (for example, by using the CSL standard input `childOps`).

This section includes the following subsections:

- Context of Evaluation
- Example

Context of Evaluation

Process taxonomy modules are evaluated for each feasible process in the context of a particular stock selection, process routing, and machine selection. A routing's process taxonomy modules are evaluated after costing of all the routing's descendent operations.

Operation taxonomy modules are evaluated (if present) for each feasible operation in the context of a particular stock selection, process routing, machine selection, GCD, operation sequence, and tool selection. Costing proceeds in a bottom-up fashion; an operation's taxonomy module is evaluated after costing of all the operation's descendant operations. That way, an operation taxonomy module can access the results of the modules associated with its children in the process-operation hierarchy, in order, for example, to aggregate their costs.

Example

Below is a taxonomy module portion for the Bar&Tube operation that creates a simple hole with a tube laser. This portion calculates the output `cycleTime`, the time required for creation of a simple hole. The main formula sums three quantities:

- `holeRapidTraverseTime`: Time required for positioning the part so that the machine's laser is directed at a point along the desired location for the hole's perimeter. Calculated based on the average distance between holes and the current machine's feed rate (`machine.tubeAxialBarFeedRate`).
- `holePierceTime`: Time required for the initial penetration of the part through the stock's entire thickness. Obtained from a lookup table, based on material, machine power, and part thickness. If the part is thicker than maximum thickness handled by the lookup table, a failure message is issued, and the table's longest pierce time is used.
- `holeCutTime`: Cutting along the perimeter of the hole, starting and ending at the pierce location. Calculated based on the hole circumference and machine feed rate. The machine's small feature feed rate is used for holes that are considered small features as determined by machine attributes and hole radius; the machine's large feature feed rate is used otherwise.

Here is the code:

```
cycleTime = cycleTimePerOp

cycleTimePerOp = holeRapidTraverseTime + holePierceTime +
  holeCutTime //secs

holeRapidTraverseTime = _
  (averageDistanceBetweenHoles / machine.tubeAxialBarFeedRate)
  //secs
```

```

averageDistanceBetweenHoles = divZero( part.minStockLength, numHoles
)

numHoles = _
  select count(a) from part.childArtifacts a where _
    a.artifactType.name=='SimpleHole' or _
      a.artifactType.name=='ComplexHole' or _
        a.artifactType.name=='MultiStepHole'

holePierceTime = pierceTime

smallFeatureRadius = max( _
  machine.smallFeatureFeedRadius, _
  machine.smallFeatureThicknessRatio * part.crossSection.thickness
)

perimeter = gcd.perimeter
holeRadius = gcd.diameter / 2

holeCutTime = { _
  (perimeter * secsPerMin / feedRateLargeFeatures) if _
    (holeRadius > smallFeatureRadius)
  (perimeter * secsPerMin / feedRateSmallFeatures) otherwise _
}

// Get pierce time depending on material composition and type
pierceTime = { _
  cutRateEntry.pierceTime if cutRateEntry != null
  fail(msg(FLT, 'Cutting rate not available for the selected
material, _
  ', material.name, _
  ' , in tubeLaserCutting table.', FRT)) _
  if cutRateEntry == null and useSlowestEntry == null
  useSlowestEntry.pierceTime otherwise _
} //secs units

```

The code uses the following predefined functions:

- `divZero`: Returns the quotient of the arguments; return 0 if the second argument is 0.

- `max`: Returns the larger of the arguments.
- `fail`: Halts evaluation, causes costing to fail, and issues a failure message to the aPriori message pane.

The code uses the following auxiliary formulas:

- `numHoles`, the number of holes (used together with `part.minStockLength` to calculate the average distance between holes) is determined with a query that counts the number of each type of hole (not just simple holes).
- `smallFeatureRadius` (used to determine the cutoff between large and small features) is the larger of the following:
 - Machine attribute `smallFeatureRadius`
 - Product of the machine attribute `smallFeatureThicknessRatio` and the `thickness` attribute of the part cross section

Code not shown above uses the lookup table `tubeLaserCutting` to retrieve the table entries `cutRateEntry` and `useSlowestEntry`.

In order to access this operation's cycle time results (along with the times for other operations), the laser cutting process taxonomy file uses the following formula:

```
sumOpsCycleTime = select sum(operation.cycleTime) from childOps
operation
```

Here, `childOps` is a CSL standard input whose value is the collection the current process's children in the process operation hierarchy. Each element of the collection has a field for each output of the element's associated CSL modules. The query designates the sum of the cycle times calculated by each child operation's taxonomy module.

Working with Zero-or-More Nodes

In some cases, a cost model requires an operation node to have multiple occurrences, where the number of occurrences is determined by cost model logic. There are two ways to implement this logic:

- With a zero-or-more template node (see [Working with Templates](#)) and an associated `zeroOrMoreOperation` Module
- With a zero-or-more template node and an associated occurrence generator Java plugin

The first approach is the simpler one: add a CSL module named **zeroOrMoreOperation** to the node. The CSL must contain a formula named **numOperations** which gets evaluated in order to determine the number of occurrences of the node.

Some existing cost models use a node occurrence generator instead of a `zeroOrMoreOperations` module. A generator is a Java plugin that provides the logic to create occurrences of zero-or-more nodes. The plugin may, in turn, call additional CSL modules to provide configurability, evaluating, for example, CSL-based operation precedence and/or compatibility rules in order to group operations under node instances. Sheet Metal – Transfer Die, for example, uses a generator to group operations by die station; the generator produces one occurrence of the parent operation Die Station for

each distinct die station that the part requires. Similarly, Machining uses a generator to group operations by setup axis, and generates one occurrence of the parent operation Setup for each group.

The occurrence generator plugin is specified by the node attribute **generatorName** (see [Working with Node Attributes](#)). Any precedence and compatibility rules are specified in CSL modules, for example, `operationPrecedence`, `operationTypePrecedence`, and `operationCompatibility` modules, though the exact set of modules supported is specific to a given plugin. These modules have standard inputs for an ordered pair of operations, and contain rules that evaluate to true if, for example, the first operation in the pair must precede the second (precedence rules), or if both operations may be grouped on the same node instance (compatibility rules).

(Note that for `operationCompatibility` modules, you can improve performance by disabling the grouping of compatible operations under circumstances in which such grouping is not needed. To do this, add an `ignoreOperationCompatibilityCSL` formula to the `operationCompatibility` module, and have it evaluate to true when compatibility grouping is unnecessary.)

If both a `zeroOrMoreOperations` module and an occurrence generator are associated with a zero-or-more node, the cost engine uses the generator. If neither are present, the cost engine generates a single occurrence of the node, and issues a warning in the log:

```
No csl found for ZeroOrMoreNode <node-name>. Creating 1 instance of node
```

The following instance generators are used by current and past cost models:

- `com.fbc.businessmodel.costing.machining.SetupAllocator`: used to allocate operations to machining setups; groups operations based on their setup axes then runs compatibility CSL to determine any further subdivision of these groupings.
- `com.fbc.businessmodel.costing.machining.SetupAllocator2`: minor variant of `SetupAllocator`; allows child operations of a multi-step hole to be assigned to different setups.
- `com.fbc.businessmodel.costing.tree.eval.StageGenerator`: generic stage generator; calls precedence CSL to group operations based on those that must be performed before others, then calls compatibility CSL to further subdivide these groups.
- `com.fbc.businessmodel.costing.tree.eval.StageGenerator2`: minor variant of `StageGenerator`; evaluation of operation precedence ignores implied ordering based on operation sequences.
- `com.fbc.businessmodel.costing.tree.eval.StageGeneratorProcess`: variant of `StageGenerator2`; considers the top-level processes rather than individual operations. (Not currently in use in starting point cost models.)
- `com.fbc.businessmodel.costing.tree.eval.DieStationGenerator`: variant of `StageGenerator` used for Progress and Transfer Die; supports injection of idle stations via `idleStationCount` CSL.
- `com.fbc.businessmodel.costing.tree.eval.DieStationGenerator2`: minor variant of `DieStationGenerator`; evaluation of operation precedence ignores implied ordering based on operation sequences as per `StageGenerator2`. (Not currently in use in starting point cost models.)

Contact aPiori Professional Services for assistance if you think you need to modify a cost model's generator-implemented zero-or-more behavior.

Working with Templates

Every routing specifies a sequence of processes or operations. A process routing specifies a sequence of processes that can be used to produce a part. An operation sequence specifies a sequence of operations that can be used to produce a given type of GCD. Process routings and operation sequences have one node for each process or operation; they can also have additional nodes, *branch nodes*, which stand for subsequences consisting of processes or operations.

A template defines a *collection* of alternative process routings or operation sequences. As with process routings and operation sequences, process and operation templates have nodes that represent processes or operations, and have branch nodes that represent subsequences consisting of processes or operations. But templates also have another type of branch node with a different purpose—to represent a group of processes or operations that are *alternatives* to one another. In this way, a template represents a set of alternative process routings or operation sequences.

This section has the following subsections:

- [About Templates](#)
- [Viewing and Editing Templates](#)

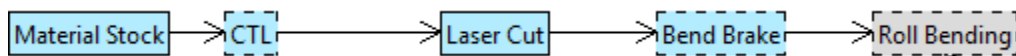
About Templates

Every CSL module is either a global module or is associated with a node in a process or operation template. Branch nodes that represent subsequences are useful because they allow association of pruning, optionality, or feasibility conditions with process or operation subsequences (as opposed to individual processes or operations).

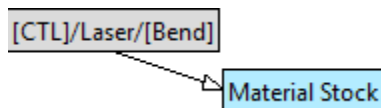
Templates are represented graphically as a set of boxes connected by arrows. (A template's graphical representation appears in the CMWB in the bottom part of the editing pane. You may have to drag the top border of the Info Panel up from near the bottom of the editing pane in order to make the Info Panel contents visible.)

Templates use three kinds of arrows:

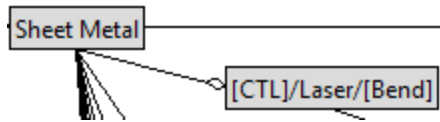
- Arrows with v-shaped arrow heads indicate order. Each such arrow points from one node to its immediate successor node in the sequence.



- An arrow with a triangular arrow head connects a branch node to the subsequence it represents. Each such arrow points from the branch node to the first member of the subsequence.



- An arrow with a diamond-shaped arrow head connects a branch node to one of the alternatives it stands for. For each alternative associated with a branch node, there is one such arrow from the branch node to that alternative.



Branch nodes are shown in grey. Process or operation nodes are shown in blue. Optional nodes appear with a dashed outline.

Templates are represented textually as follows:

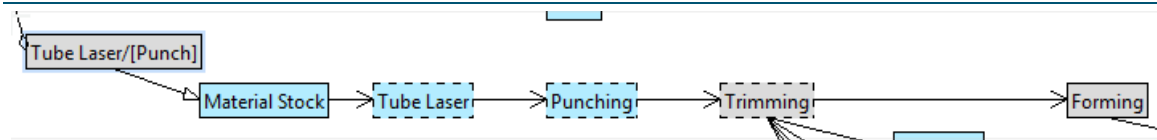
- Each node has a name, the name displayed under **Processes, GCDs, and Operations** in the CMWB navigation pane. Names that contain a space must be enclosed in single quotes. No two nodes in the same template should have the same name.

Since a name might not be unique across a process group (and its associated secondary process groups), it is sometimes necessary to use a *pathname*. Node names appear in the **Node Attributes** tab of the editing pane for a given process, operation, or branch node. Pathnames are colon-separated lists of names, consisting of the node name followed by its ancestors in the **Processes, GCDs, and Operations** hierarchy, and ending with the name of the process group. Each name in the path must be enclosed in single quotes if it contains a space.

- Node names that are separated by spaces indicate order. The graphical equivalent is a connection made by an arrow with a v-shaped arrow head.
- Each template has a top-level branch node.
- A node name followed by ::= indicates a branch node. If ::= is followed by a space-separated sequence of node names, the branch node stands for a sub-sequence. The graphical equivalent is a connection from the branch node to the first node in the sequence by an arrow with a triangular arrow head.
- A node name followed by ::=, followed by a vertical-bar-separated (pipe-separated) sequence of node names indicates a branch node that stands for a group of alternatives. The graphical equivalent is a group of arrows with diamond-shaped arrow heads, with an arrow from the branch node to each alternative.
- A node name preceded by a double vertical bar indicates an alternative that must be user-selected in order to be included in routings generated from the template. If the alternative is not explicitly selected by the end user, the cost engine does not include the alternative in generated routings.
- Names of optional nodes are enclosed in square brackets when they appear to the right of ::= (but not when they appear to the left of ::=).
- A node name followed by a star, *, indicates a zero-or-more node—see [Working with Zero-or-More Nodes](#).
- Lines that start with # indicate template portions that are not visible to the end user in aPriori.

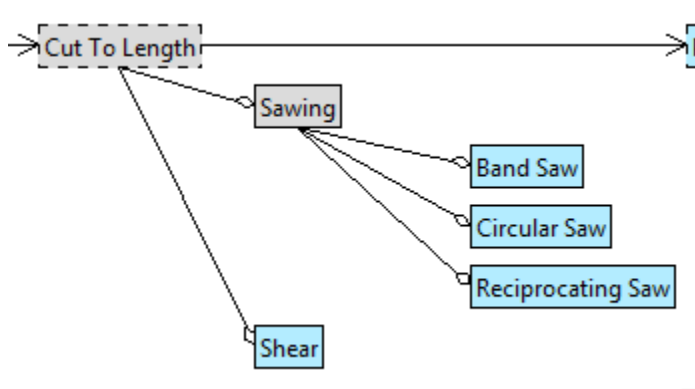
Here are some examples of template specification text, together with their corresponding graphical representations.

```
#'Tube Laser/[Punch]' ::= 'Material Stock':'Bar & Tube Fab' _
    ['Tube Laser':'Bar & Tube Fab'] [Punching:'Bar & Tube Fab'] _
        [Trimming] Forming
```



```
'Cut To Length' ::= Sawing || Shear:'Bar & Tube Fab'
```

```
Sawing ::= 'Band Saw':'Bar & Tube Fab' || 'Circular Saw':'Bar & Tube Fab' || _
'Reciprocating Saw':'Bar & Tube Fab'
```



For more examples, see [Adding New Processes and Operations to Templates in Common Task Examples](#).

Viewing and Editing Templates

To view and edit a template for a given GCD type, follow these steps:

- 1 In the CMWB navigation pane, expand the **Templates** node.
- 2 Double click the desired GCD type. The template text appears in the editing pane; the template's graphical representation appears below it, in the **Template Graph** tab. (You may have to drag the top border of the Info Panel up from near the bottom of the editing pane in order to make the Info Panel contents visible.)

Note that process-level templates are associated with the component GCD type.

- 3 To edit the template, select **Override Object** from the CMWB **Edit** menu. You can now modify the text.
- 4 Select **Save** from the **File** menu to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu.

4 Cost Engine Details

This chapter provides details on the cost engine algorithm, focusing on the flow of evaluation of the various types of CSL modules.

The following topics are covered here:

- Hierarchies
 - Algorithm Overview
 - Process Template Expansion and Pruning
 - Material Stock Selection
 - Feasibility and Machine Selection
 - Operation Assignment
 - Operation Costing
 - Process Costing
-

Hierarchies

The cost engine algorithm employs a number of hierarchical (that is, tree structured) data structures. The description of the cost engine algorithm in this chapter often refers to visiting the nodes of a hierarchy or traversing a hierarchy. In order to understand the description, it's important to clearly identify and distinguish among the various types of hierarchies:

- **GCD hierarchy:** generated by GCD extraction, and represented in the aPriori interface in the **Geometric Cost Drivers** pane. The parent/child relation in this hierarchy usually represents the spatial part-of relation. When the cost engine considers a particular process routing (or process routing/material stock combination), it descends the current part's GCD hierarchy, choosing an operation sequence for each GCD it visits along the way, and additionally establishing the process-operation hierarchy (see below) by finding a parent process or parent operation for each operation in the chosen sequence (see [Operation Assignment](#)).
- **Templates:** defined by the cost model to provide a compact representation of a variety of candidate process routings or operation sequences. Note that a template is different from the routings or sequences that instantiate the template. The cost engine traverses a template during expansion in order to generate instantiations, evaluating template pruning modules along the way (see [Process Template Expansion and Pruning](#)).
- **Template instantiations:** generated by template expansion. Each template instantiation is particular process routing or operation sequence. The cost engine performs a traversal of each candidate process routing in order to evaluate each node's optionality module; it also performs a traversal of each candidate process routing in order to evaluate feasibility and select a machine for each node; and it performs a traversal of each candidate process routing in order cost each process (see [Feasibility and Machine Selection](#) and [Process Costing](#)). Similarly, the cost engine performs traversals of operation sequences in order to assign parents, evaluate feasibility, select tools, and perform operation costing (see [Operation Assignment](#) and [Process and Operation Taxonomy](#)).
- **Process-GCD-operation hierarchy:** represented by the **Processes, GCDs & Operations** tree in the CMWB navigation pane. This hierarchy determines each operation's associated GCD type, as well as each process and operation's potential or permissible children in the process-operation hierarchies—see below. This hierarchy (together with the current process routing or parent operation sequence) constrains the assignment of parents in the process-operation hierarchies (see [Operation Assignment](#)).
- **Process-operation hierarchies:** generated during descent of the GCD hierarchy—see GCD hierarchy, above. The parent/child relation in this hierarchy usually represents the temporal part-of relation (that is, a child operation is a part of the process or operation represented by the parent, where the child takes up only a part of the time required for the parent procedure). When the cost engine considers a particular process routing (or process routing/material stock combination), before it evaluates the processes' process taxonomy modules, it costs the operations required to create the current part's GCDs as part of these processes. When the cost engine costs these operations, it ascends (using post-

order, depth first tree traversal) the process-operation hierarchies, evaluating each operation taxonomy module along the way (see [Process and Operation Taxonomy](#)).

Algorithm Overview

In very general terms, the cost engine algorithm costs a variety of alternative process routings for a given part, and picks the one with the lowest cost. The candidate routings are derived from process templates and template pruning CSL modules (see [Process Template Expansion and Pruning](#)). For process groups that support material stocks, the cost engine actually costs a variety of routing/ stock combinations, and picks the best one. It derives the candidate stocks from the material stock selection CSL module (see [Material Stock Selection](#)).

For each candidate process routing (or process routing/material stock combination), the algorithm, broadly speaking, proceeds in four stages:

- 1 Feasibility and machine selection:** Evaluates the routing's feasibility. Eliminates the routing from consideration if it is infeasible; otherwise, picks a machine for each feasible node of the routing. See [Feasibility and Machine Selection](#) for details.
- 2 Operation assignment:** Descends the GCD hierarchy, choosing an operation sequence for each GCD, and additionally finding a parent process or parent operation for each operation in the chosen sequence (thus establishing the process-operation hierarchy), as well as finding a tool for each operation in the chosen sequence that requires one. See [Operation Assignment](#) for details.
- 3 Operation costing:** Ascends the process-operation hierarchy, evaluating the cost of each operation along the way (possibly relying on the results of evaluating the cost of descendent operations). See [Process and Operation Taxonomy](#) for details.
- 4 Process costing:** Traverses the process routing, evaluating the cost of each process along the way (possibly relying on the results of evaluating operation costs). The cost engine then aggregates the costs across processes, yielding a fully-burdened, per-part cost. See [Process Costing](#) for details.

Once each routing (or routing/stock combination) has been considered, the cost engine selects the one with the lowest fully-burdened cost. If all candidate routing/stock combinations have been eliminated from consideration, the part fails to cost.

Pseudo code for this high-level algorithm might take the following form:

```
Template Pruning
for each pruned, licensed process routing {

    Material Stock Selection
    for each selected material stock {

        Stage 1: Feasibility and Machine Selection
        Stage 2: Operation Assignment
```

```
        Stage 3: Operation Costing
        Stage 4: Process Costing
    }

}

Select Optimal Routing and Stock
```

Process Template Expansion and Pruning

Once GCD extraction has been performed, the cost engine begins costing a part by expanding the process-level templates associated with the current process group. Expansion potentially produces every instantiation of the templates, that is, every possible process routing for the process group. Each routing includes some, but not all, nodes from a template.

During expansion, the cost engine visits each template node, and evaluates the node's associated template pruning module, if there is one. If a node's template pruning module returns false, the node is removed from the template. All the node's descendents in the template are removed as well. This can prevent the production of a large number irrelevant routings.

A template pruning rule might, for example, eliminate all turning routings for parts that have no valid turning axis.

Material Stock Selection

For process groups that use material stock, the complete list of available stocks (for the current user-selected material) is presented in aPriori's **Material Selection** dialog, when the **Material Stock** pane is expanded. Based on the geometry of the current part, the cost engine can narrow down the stocks it considers by evaluating a material stock selection module.

If aPriori's material stock mode is **Auto** rather than **Manual**, then for each process routing that the cost engine considers, it checks the routing's nodes for the presence of the node attribute **utilizationProcess**.

- If there is exactly one node such that (1) the node has the **utilizationProcess** attribute and (2) this attribute has a value, then the cost engine evaluates the stock selection module associated with the node named by the attribute value. (No stock selector is evaluated if the value does not name a node with a stock selector.)
- If there is more than one node such that (1) it has the **utilizationProcess** attribute and (2) this attribute has a value, then the cost engine evaluates the stock selection module associated with the node named by the value of the left-most of these nodes. (No stock selector is evaluated if the value does not name a node with a stock selector.)

- Otherwise, if there is exactly one node that has the **utilizationProcess** attribute (but the attribute has no value), the cost engine evaluates the stock selection module associated with the node that has the attribute.
- Otherwise, if there is more than one node that has the **utilizationProcess** attribute (but the attributes have no values), the cost engine evaluates the stock selection module associated with the right-most active node that has the attribute.

For example, for the sheet metal process group, if a routing includes the branch node `Prog Die`, which has a node attribute **utilizationProcess** whose value is `Progressive Die`, the cost engine evaluates the stock selection module associated with `Progressive Die`.

If the stock selection module establishes a collection of stocks as the value for the CSL output `validMaterialStocks`, the cost engine considers each stock in the collection.

If the stock selection module does not establish a non-null value for the CSL output `validMaterialStocks`, the cost engine considers only one, virtual stock, specified as follows:

- For the Machining process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:
 - Length:
 - CSL output `virtualStockLength`, if non-null.
 - Otherwise, the larger of the current part's `minStockLength` and the value of the plant variable `standardStockLength`, if the plant variable is defined.
 - Otherwise, the larger of the current part's `minStockLength` and 20 feet.
 - Width:
 - CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the width of the current part's cross-section.
 - Height:
 - CSL output `virtualStockHeight`, if non-null.
 - Otherwise, the height of the current part's cross-section.
 - Inside diameter:
 - CSL output `virtualStockInsideDiameter`, if non-null.
 - Otherwise, the inside diameter of the current part's cross-section.
 - Outside diameter:
 - CSL output `virtualStockOutsideDiameter`, if non-null.
 - Otherwise, the outside diameter of the current part's cross-section.
 - Thickness:
 - CSL output `virtualStockThickness`, if non-null.
 - Otherwise, the thickness of the current part's cross-section.
- For the Sheet Metal process group, virtual stock dimensions are determined by the values of other CSL outputs for the stock selection module, the dimensions of the current part, and plant variables, as follows:
 - Length:
 - The larger of the blank's length and the CSL output `virtualStockLength`, if non-null.

- Otherwise, the larger of the blank's length and the value of the plant variable `standardStockLength`, if the plant variable is defined.
- Otherwise, the larger of the blank's length and 8 feet.
- o Width:
 - The larger of the blank's width and the CSL output `virtualStockWidth`, if non-null.
 - Otherwise, the larger of the blank's width and the value of the plant variable `standardStockWidth`, if the plant variable is defined.
 - Otherwise, the larger of the blank's width and 4 feet.

Feasibility and Machine Selection

In this stage, the cost engine evaluates various CSL modules associated with the nodes of the current process routing:

- 1 First the cost engine visits each node in the routing, evaluating each node's optionality module, if there is one. If the module returns true, the cost engine makes the node non-optional. If the module returns false, the cost engine removes the node from the current routing.
- 2 Next the cost engine again visits each node in the current routing. For each node it does the following:
 - o Evaluates the node's feasibility module. The cost engine proceeds as follows:
 - If the module returns false and the node is non-optional, the current routing as a whole is deemed infeasible, and the cost engine moves on to consideration of a new process routing.
 - If the module returns false and the node is optional, the current node (together with any descendants) is removed from the routing, and the cost engine proceeds to the next node of the current routing, if there is one.
 - If the module returns true or if there is no feasibility module, the cost engine proceeds to machine selection.
 - o Attempts to select a machine for the current process (except when the node's **selectMachineAfterOpAssignment** attribute is non-null—see [Process Costing](#))—and except when a machine has already been selected for an ancestor of the current node in the current routing). aPriori has four machine selection modes (see the **Edit Machine Selection** dialog in aPriori):
 - **aP Select:** The cost engine evaluates the node's machine selection module, putting into scope the CSL input `machines`, and establishing its value as a collection with one machine for each line of the aPriori machine table (see the **Edit Machine Selection** dialog in aPriori). If the machine selection module fails to establish a non-null value for the CSL output `machine`, the current routing is deemed infeasible, and the cost engine moves on to consideration of a new process routing. Otherwise, the value of `machine` is assigned to the current process.
 - **User Select/ if not feasible, fail to cost:** The cost engine evaluates the node's machine selection module, putting into scope the CSL input `machines`, and establishing its value as a collection whose only element is the user-selected machine. If the machine selection module fails to establish a non-null value

for the CSL output `machine`, the current routing is deemed infeasible, and the cost engine moves on to consideration of a new process routing. Otherwise, the value of `machine` is assigned to the current process.

- **User Select/ if not feasible, auto-select:** The cost engine evaluates the node's machine selection module, putting into scope the CSL input `machines`, and establishing its value as a collection whose only element is the user-selected machine. If the machine selection module fails to establish a non-null value for the CSL output `machine`, the cost engine proceeds as described for **aP Select**, above. Otherwise, the user-selected machine is assigned to the current process.
- **User Select/do not check feasibility:** The cost engine assigns the user-selected machine to the current process.

In summary, this stage involves the evaluation of the following types of CSL modules:

- Process optionality
- Process feasibility
- Machine selection

These types of modules are evaluated in the context of a process routing (and possibly a material stock) and a particular routing node, but operation sequences have not yet been chosen, and no process has yet been costed. Throughout the evaluation of optionality and feasibility modules, no machine has yet been selected. See Algorithm Overview.

Operation Assignment

In this stage, the cost engine visits each node in the current part's GCD hierarchy, attempting to assign it an operation sequence. For each GCD in the hierarchy, the cost engine picks an operation sequence that is associated (via operation templates) with that GCD's type.

For each GCD, the cost engine expands its associated operation templates. It then considers several alternative sequences if necessary, and selects the first one that is successful in the following sense:

- All the operations in the sequence can be assigned a parent process or operation.
- All the sequence's non-optional operations are feasible and can be assigned a tool (if appropriate)
- All children (if there are any) of the current GCD can successfully be assigned an operation sequence in the sense described here.

Note that the cost engine does *not* compare the costs of different candidate operation sequences as it does with candidate process routings; instead, it simply selects the first sequence that is successful in the above sense.

If no successful sequence is found, the GCD fails to cost, and the cost engine issues a warning to aPriori.

To make the assignments, the cost engine performs the following recursive procedure on each child GCD of the current part's component GCD:

Expand the associated operation templates, and for each resulting operation sequence, do the following:

- 1 Parent assignment: Find an assignment of operations in the current sequence to processes in the current process routing (or to operations in the sequence currently assigned to the parent GCD of the current GCD) such that both the following hold:
 - o Each operation is an allowable child of its assigned process or operation (see the navigation pane of the CMWB).
 - o If one operation is earlier in the sequence than another, the first operation is assigned a parent that is the same or earlier than (in the process routing) the parent assigned to the other operation.

If no such assignment is possible, the cost engine issues an error message to the aPriori message tree, and moves on to the next operation sequence.

- 2 Feasibility and tool selection:
 - o For each node in the current operation sequence, evaluate the node's optionality module.
 - o For each node in the current operation sequence:
 - Evaluate the node's feasibility module. If the module returns false and the node is non-optional, the current sequence as a whole is deemed infeasible, and the cost engine moves on to consideration of a new operation sequence. If the module returns false and the node is optional, the current node (together with any descendents) is removed from the sequence, and the cost engine proceeds to the next node of the current sequence, if there is one. If the module returns true or if there is no feasibility module, the cost engine proceeds to tool selection.
 - Evaluate the node's tool selection module. If the module fails to establish a non-null value for the CSL output `tool`, the current sequence is deemed infeasible, and the cost engine moves on to consideration of a new operation sequence. Otherwise, the value of `tool` is assigned to the current operation.
- 3 Child GCD operation assignment: Perform these steps (1, 2, and 3) recursively for the child GCDs (if there are any) of the current GCD. If the procedure succeeds, the cost engine selects the current operation sequence for the current GCD. If the procedure fails, the cost engine moves on to consideration of a new operation sequence.

Pseudo code for the procedure performed on each GCD might take the following form:

```
boolean assignOperations(GCD) {
    Expand operation templates for GCD
    for each operation sequence associated with the GCD {

        Parent Assignment
        if parent assignment was unsuccessful
            continue (move on to the current GCD's next
            operation sequence)
```

```

Feasibility and Tool Selection
if feasibility and tool selection were unsuccessful
    continue (move on to the current GCD's next
operation sequence)

for each child GCD of GCD {
    success = assignOperations(child GCD) //
recursion
    if !success break;
}
if !success
    continue (move on to the current GCD's next
operation sequence)
else {
    assign the current sequence to the current GCD
    return true
}
}
return false
}

```

For process groups that have templates with zero-or-more nodes (see [Working with Templates](#), there is an additional step immediately before feasibility and tool selection (step 2, above, in this section). The cost engine evaluates a number of modules (such as precedence and compatibility modules), if present, in order to expand these nodes into zero or more individual operations.

This procedure is performed on the GCDs by performing up to four passes on the GCD hierarchy. This allows cost models such as machining to select operation sequences that minimize the number of different setup axes required to manufacture the current part. Pseudo code for these passes might take the following form:

```

// pass 1: CSL input passNumber has value 1
foreach non-axisymmetric child gcd of component {
    assignOperations(gcd)
}

// pass 2: CSL input passNumber has value 2
foreach axisymmetric child gcd of component {
    assignOperations(gcd)
}

// pass 3: CSL input passNumber has value 3

```

```
foreach unassigned non-axisymmetric child gcd of component {
  assignOperations(gcd)
}

// pass 4: CSL input passNumber has value 4
foreach unassigned non-axisymmetric child gcd of component {
  assignOperations(gcd)
}
```

During each pass, some operations are treated as if they are infeasible, so their consideration can be deferred to a later pass:

- Pass 1: Only operation-sequence nodes with an ancestor whose value for the node attribute `assignOpsWithActivePrimaryAxesFirst` is `true` are treated as potentially feasible; all others are treated as infeasible.
- Pass 2: All hole-related and axigroove-related operations are treated as potentially feasible; all others are treated as infeasible.
- Pass 3: All surface-related operations are treated as potentially feasible; all others are treated as infeasible.
- Pass 4: Only nodes with an ancestor whose value for the node attribute `runInFourthCostingPass` is `true` are treated as potentially feasible; all others are treated as infeasible.

In addition, for all passes, operation feasibility modules can control which operations are considered feasible in a given pass by including rules that refer to the CSL input `passNumber`.

For passes 3 and 4, the GCDs are visited in order of descending size (based on the `areaFinished` GCD attribute), if some node of the current process routing has value `true` for the node attribute `sortGcdsByFinishArea`.

In summary, this stage involves evaluation of the following module types:

- Operation optionality
- Operation feasibility
- Tool selection

These types of modules are evaluated in the context of a process routing (and possibly a material stock), an assignment of machines to processes in the current routing, and a candidate operation sequence for the current GCD and all its ancestors. There are no candidate operation sequences for descendants of the current GCD. During any given evaluation, operation sequences for the current process routing have not yet been finalized, and no process has yet been costed.

Operation Costing

In this stage, the cost engine traverses each process-operation hierarchy. It uses post-order, depth first tree traversal, evaluating each operation's taxonomy module. This means, essentially, that costing proceeds in a bottom-up fashion, so that all the children of a given operation are costed before that operation is costed. That way, an operation

taxonomy module can access the results of the modules associated with its children in the process-operation hierarchy, in order, for example, to aggregate their cycle times.

Process Costing

Process costing proceeds in two passes: a right-to-left pass (the yield pass) and a left-to-right pass.

In the first pass, the yield pass, the cost engine visits each node in the process routing. For each node, the cost engine does the following:

- If the node's **selectMachineAfterOpAssignment** attribute is non-null (and so machine selection was not performed during Stage 1), the cost engine performs machine selection as described in step 2 of [Feasibility and Machine Selection](#).
- The cost engine evaluates only the following formulas (if present) in each node's process taxonomy module:
 - finishMass
 - stripNestingPitch
 - roughLength
 - numPartsPerSheet
 - utilizationWithAddendum
 - scrapMass
 - roughMass
 - utilizationWithoutAddendum
 - utilization
 - materialYield
 - goodPartYield

This pass proceeds right to left; that is, nodes later in the routing are visited first.

In the second pass, the cost engine again visits each node in the process routing, and evaluates the rest of the formulas in each node's process taxonomy module. This pass proceeds left to right.

Once all processes taxonomy modules have been fully evaluated, the cost engine evaluates the Site Cost Model taxonomy module, which typically aggregates the results of the individual process taxonomy modules, producing the fully-burdened, per-part cost as well as the aggregate costs presented in aPriori's **Cost Summary** tab.

For process groups that have templates with zero-or-more nodes, there is an additional step immediately before the yield pass. The cost engine evaluates special modules, if present, in order to expand these nodes into zero or more individual operations.

5 Cost Scripting Language Reference

This chapter provides a reference on aPriori's Cost Scripting Language (CSL). The code in a CSL module specifies the rules and formulas that make up a particular portion of a cost model (such as the portion that computes a cost taxonomy for a particular process). So experienced users can employ CSL to implement and customize the details of cost model logic. For more information on working with cost model logic, see also [Working with Cost Model Logic](#). For information on working with cost model *data*, see [Working with Cost Model Data and Metadata](#).

This chapter includes the following topics:

- Overview
 - Modules, Inputs, and Outputs
 - Imports
 - Formulas and Rules
 - Expressions
 - Identifiers and Literals
 - Comments and Line Continuation
 - Predefined Functions
 -
-

Overview

The high-level logic of any cost model is specified by the cost engine algorithm (see Cost Engine Details). This part-costing algorithm specifies the flow of evaluation of the various types of CSL modules. For example, it specifies that each process's machine selection module is evaluated only if its feasibility module evaluates to `true`. The code in a CSL module specifies the rules and formulas that make up a particular portion of a cost model, such as the portion that evaluates the feasibility of a particular process, or selects a machine for a process, or computes a process's cost taxonomy.

Module Types

The various types of CSL modules can be divided into three categories:

- *Taxonomy modules* calculate those outputs, such as `cycleTime` and `laborCost`, that are specified by the module's associated formula table in the CMWB.
- *Selection modules* establish values for special outputs associated with their module types, for example, `machine` for machine selection modules and `tool` for tool selection modules.
- *Modules that return a boolean value*, such as feasibility modules, consist primarily of rules. They generally return `true` if all the rules succeed; they return `false` if any rule fails.

Taxonomy modules include the following module types:

- **Process taxonomy:** evaluated for each feasible process in the context of a particular stock selection, process routing, and machine selection. A routing's process taxonomy modules are evaluated after costing of all the routing's descendent operations. Process taxonomy modules calculate various physical quantities, durations, and monetary quantities in order to generate a complete cost taxonomy for the current part and process.
- **Operation taxonomy:** evaluated (if present) for each feasible operation in the context of a particular stock selection, process routing, machine selection, GCD, operation sequence, and tool selection. An operation's taxonomy module is evaluated after costing of all the operation's descendant operations. Operation taxonomy modules calculate various physical quantities, durations, and monetary quantities in order to generate a complete cost taxonomy for the current GCD and operation.

Selection modules include the following module types:

- **Material stock selection:** evaluated for each pruned, process-level routing. This module (if present for some process in the current routing) narrows down the candidate material stocks.
- **Machine selection:** evaluated (if present) for each feasible process in the context of a particular stock selection and process routing. This module selects a machine for the current process.
- **Tool selection:** evaluated (if present) for each operation in the context of a particular stock selection, process routing, machine selection, GCD, and operation sequence. This module selects a tool for the current operation.

Modules that return a value include, among others, the following module types:

- **Process feasibility:** evaluated (if present) for each process in the context of a particular stock selection and process routing. If the module returns true, the process is considered feasible, and the cost engine proceeds to machine selection; if the module returns false, the current process routing is abandoned (unless the process is optional, in which case it is removed from the current process routing).
- **Operation feasibility:** evaluated (if present) for each operation in the context of a particular stock selection, process routing, machine selection, GCD, and operation sequence. If the module returns true, the operation is considered feasible, and the cost engine proceeds to tool selection; if the module returns false, the current operation sequence is abandoned (unless the operation is optional, in which case it is removed from the current operation sequence).
- **Template pruning:** evaluated (if present) during the initial expansion of the process-level templates. If the module returns true, the current node is included in the current routing; if the module returns false, the node is removed from the current routing.
- **Process optionality:** evaluated (if present) for each process in the context of a particular stock selection and process routing. If the module returns true, the current process is included in the current routing; if the module returns false, the process is removed from the current routing.
- **Operation optionality:** evaluated (if present) for each operation in the context of a particular stock selection, process routing, machine selection, GCD, and operation sequence. If the module returns true, the current operation is included in the current sequence; if the module returns false, the operation is removed from the current sequence.

See Cost Engine Details for detailed information on the flow of evaluation of the various types of CSL modules.

Module Contents

Each CSL module contains the following kinds of top-level syntactic elements:

- *Import directives* effectively include the text of a specified file in the current CSL file. See [Imports](#) for more information.
- *Formulas* are essentially named expressions. When a formula is evaluated, the expression is evaluated, and the result is established as the value of the name. The expression can evaluate to a number, string, boolean, collection, or object. See [Formulas](#) for more information.
- *Rules* are essentially named or unnamed boolean expressions. When a rule is evaluated, the expression is evaluated, and the rule returns the result. For a named rule, if the result is false, the message with a matching name is evaluated. See [Rules](#) for more information.
- *Messages* are essentially named strings. When a rule with a matching name fails, the string is appended to the cost engine log. See [Messages](#) for more information.

- *Function definitions* are essentially parameterized formulas. Function invocation expressions specify actual parameters for the definition's formal parameters. See [Function Definitions](#) for more information.
- *Set blocks* are sequences of formulas enclosed in braces. A formula that is *not* within a set block is evaluated only if either it is an output formula or its evaluation is required (directly or indirectly) for the evaluation of an output formula (see [Outputs](#)). A formula that *is* within a set block is always evaluated as part of the evaluation of the module that contains it.. Set blocks are typically used to pass information to other modules by setting fields of input objects. See [Set Blocks](#) for more information.

See [Modules, Inputs, and Outputs](#) for information on the syntactic structure of CSL modules.

Values and Expressions

CSL supports the following types of values

- **Arithmetic:** can be designated by expressions (see below), numeric literals such as `50` or `21.75`, or identifiers such as `roughMass` or `machine.cycleTime`. See [Numerical Literals](#) for more information about arithmetic values.
- **String:** can be designated by expressions (see below), string literals, such as `'Material Cost'`, or identifiers such as `op.name`. See [String Literals](#) for more information about strings.
- **Boolean:** can be designated by expressions (see below), the boolean literals `true` and `false`, or identifiers such as `op.isManualOverride`.
- **Object:** can be designated by expressions (see below) or identifiers such as `part`, `gcd`, or `machine`. Each object has one or more named fields which you can access using dot notation, as in `part.volume` or `part.material`. The value of a field can be any type of CSL-supported value, including a collection or other object.
- **Collection:** can be designated by expressions (see below) or identifiers such as `childOps` or `tubLaserCutting`. Note that a cost model's lookup tables are collections; each collection element is a table row, and each collection-element field corresponds to a table column. You can retrieve a collection's elements by using `query` or `foreach` expressions.

See [Identifiers and Literals](#) for information on CSL literals and identifiers.

CSL supports the following kinds of expressions:

- **Arithmetic:** can be formed using binary and unary arithmetic operators, such as `*` and `-`. See [Arithmetic Expressions](#) for more information.
- **String:** can be formed using the string concatenation operator `+`. See [String Expressions](#) for more information.
- **Boolean:** can be formed using unary and binary logical operators such as `&&` and `~`, as well as binary arithmetic comparison operators such as `==` and `<=`. See [Boolean Expressions](#) for more information.

- **Conditional:** evaluates to one of several alternative values, depending on the value of boolean expressions associated with the alternative values. See [Conditional Expressions](#) for more information.
- **Function invocation:** evaluates to the result of substituting the invocation's actual parameters for the formal parameters in the corresponding function definition. See [Function Invocations](#) for more information.
- **Query:** retrieves or aggregates collection elements or values. These expressions have essentially the same semantics as SQL queries. See [Query Expressions](#) for more information.
- **Foreach:** evaluates to the result of applying a formula to each collection element in turn. See [Foreach Expressions](#) for more information.

In general, a type mismatch (for example, using a string as an operand of a boolean operator) causes an exception (that is, runtime error) in CSL. In some cases, the cost engine converts an arithmetic value to a string in order to resolve a mismatch. Similarly, `true` and `false` are converted to 1 and 0 (respectively) in some cases.

Modules, Inputs, and Outputs

Every CSL module has an associated set of *input identifiers* and an associated set of *output identifiers*. Some CSL modules return a boolean value. When the cost engine evaluates a CSL module, it does the following:

- Establishes values for all the input identifiers. See [Inputs](#) for detailed information on inputs.
- *Imports* any explicitly specified import modules, as well as certain CSL modules global to the current process group. See [Imports](#) for information on imports.
- Evaluates the module's *rules* and *formulas*. The flow of evaluation depends on the module's type (see below). Formula evaluation establishes values for output identifiers. See [Outputs](#) for detailed information on outputs. Rule evaluation returns boolean values, and can cause the module to return a boolean. See [Return Values](#) for more information on return values.
- Evaluates each formula in a *set block*. See [Set Blocks](#) for information on set blocks.

The following summarizes the flow of evaluation for formulas and rules:

- For taxonomy and selection modules, the cost engine generally evaluates all the *output formulas*, that is, formulas whose left hand side is an output identifier. This establishes values for the output identifiers. Other formulas and rules are evaluated only if required for the evaluation of some output formula. See [Formulas](#) for more information on formula evaluation.
- For modules that return true or false, rules are evaluated in the order in which they appear in the module. If a rule fails, rule evaluation terminates, and module evaluation returns `false`; if all the rules succeed, module evaluation returns `true`. Formulas are evaluated when required for rule evaluation. For feasibility modules, after rule evaluation terminates and before the module returns a value, the cost engine evaluates output formulas, if there are any. See [Rules](#) for more information on rules.

Note that an instance of any of these syntactic categories can span lines only through the use of line continuations (with certain exceptions)—see [Comments and Line Continuation](#). Note also that CSL supports comments. See [Comments and Line Continuation](#) for more information.

Syntax

Each CSL module has the following syntax:

```
[<import-directive>*]
{<formula> | <rule> | <message> | <function-definition> | <set-block>}[
<formula> | <rule> | <message> | <function-definition> | <set-
block>}]*
```

See [Imports and Formulas and Rules](#) for information on each syntactic category mentioned above:

Inputs

Input values include data derived from the current VPE and the current part. Input values also sometimes include certain outputs of other CSL modules. When the cost engine evaluates a module, it puts into scope each input identifier, that is, it establishes values for the module's input identifiers.

Note that you can view input identifiers and their values by setting a breakpoint in the CSL debugger. Click the **Inputs/Formula Results** tab when the debugger stops at a breakpoint. See [Using the CSL Debugger](#) in [Working with Cost Model Logic](#) for information on setting a breakpoint.

Note also that you can test whether an identifier is in scope with the predefined function `safeEval`--see [Miscellaneous Functions](#). Other uses of an identifier that is not in scope cause an exception.

The input identifiers listed below are associated with every type of CSL module, unless otherwise specified.

Accessibility

The value of this identifier is an enumeration object with the following fields:

- OBSCURED
- SIDE_A
- SIDE_B
- THROUGH
- UNDERCUT
- UNKNOWN

allOps

The value of this identifier is a collection that has one element for each already-costed operation of the current routing. Each element is an object that has one field for each already-evaluated output formula of a module associated with the operation, as well as one field for each attribute of the operation's associated GCD.

You can view a list of these formulas and attributes as follows:

- Formula names:
 - In CMWB, select the operation or process and click the **CSL Modules** tab.
 - See Outputs for information on standard, implicit output formulas for each CSL module type.
- GCD attributes: in aPriori, select a GCD in the **Geometric Cost Drivers** pane. The `gcd` object field names are the same as the names displayed in the **Name** column of the **Geometric Cost Drivers** pane, modified to remove spaces and to make the initial character lower case.

Each element of `allOps` also has the following fields:

- `childOps`: collection with an element for each already-costed child operation of the child operation. This allows recursive access to descendants of the current process or operation.
- `displayName`: name of the operation as translated into the local language.
- `ignoreFeasibility`: `true` if the current routing node was explicitly specified by the user to be included in the current routing regardless of feasibility; `false` otherwise.
- `isManualOverride`: `true` if the current routing node was explicitly selected by the user to be included in the current routing; the value is `false` otherwise.

`allOps` is in scope for all types of CSL modules, *except* the following:

- Operation precedence
- Operation type precedence
- Compatibility
- Idle station count

BarShape

The value of this identifier is an enumeration object with the following fields:

- `ROUND_BAR`
- `SQUARE_BAR`
- `RECTANGULAR_BAR`
- `ROUND_TUBE`
- `SQUARE_TUBE`
- `RECTANGULAR_TUBE`
- `CHANNEL_BAR`
- `ANGLE_BAR`
- `HEX_BAR`
- `I_BEAM`
- `UNKNOWN`

childOps

The value of this identifier is a collection that has one element for each already-costed child operation of the current process or operation. Each element is an object that has one field for each output formula of a module associated with the operation, as well as one field for each attribute of the operation's associated GCD.

You can view a list of these formulas and attributes as follows:

- Formula names:
 - In CMWB, select the operation or process and click the **CSL Modules** tab.
 - See Outputs for information on standard, implicit output formulas for each CSL module type.
- GCD attributes: in aPriori, select a GCD in the **Geometric Cost Drivers** pane. The `gcd` object field names are the same as the names displayed in the **Name** column of the **Geometric Cost Drivers** pane, modified to remove spaces and to make the initial character lower case.

Each element of `childOps` also has the following fields:

- `childOps`: collection with an element for each already-costed child operation of the child operation. This allows recursive access to descendants of the current process or operation.
- `displayName`: name of the operation as translated into the local language.
- `ignoreFeasibility`: `true` if the current routing node was explicitly specified by the user to be included in the current routing regardless of feasibility; `false` otherwise.
- `isManualOverride`: `true` if the current routing node was explicitly selected by the user to be included in the current routing; the value is `false` otherwise.
- `name`: name of the operation as specified in the associated template.

`childOps` is in scope for all types of CSL modules, *except* the following:

- Operation precedence
- Operation type precedence
- Compatibility
- Idle station count

childResults

The value of this identifier is a collection with one element for each child process node of the current node. Each child node's associated object has one field for each already-evaluated output formula of the child node. Other fields include the following:

- `name`
- `simpleName`

ComplexHoleSubType

The value of this identifier is an enumeration object with the following fields:

- `RECTANGULAR_ROUNDED`

- OBROUND
- RECTANGULAR
- UNKNOWN

constants

The value of this identifier is an object that has one field for each CSL predefined constant for the current process group. To view a complete list of predefined constants, click **Globally Available CSL/Lookup Tables** under **Global Cost Model Information** in the navigation pane of the CMWB, select the **Modules** tab, and click the **constants** folder.

defaultMachine

The value of this identifier is an object that represent the default machine for the current process. It has one field for each attribute of the default machine. You can view machine attributes for a given process in the **Edit Machine Selection** dialog (right click a process name in aPriori's **Manufacturing Process** pane, and select **Machine Selection**). Note that this is in scope only if there is a current process; is not in scope for template pruning or material stock selection.

Direction

The value of this identifier is an enumeration object with the following fields:

- UP
- DOWN
- BOTH
- EITHER
- UNKNOWN

EdgeShape

The value of this identifier is an enumeration object with the following fields:

- CHAMFER
- ROUND
- SHARP
- UNKNOWN

FormType

The value of this identifier is an enumeration object with the following fields:

- TAB
- BRIDGE
- EMBOSS
- STAMP
- GUSSET
- CURVED_BEND
- SIDE_ACTION

- EXPANSION
- REDUCTION
- UNKNOWN

gcd

This is an object that has one field for each attribute of the current GCD. The field names and values are the same as the attribute names and values. You can view a list of these attributes as follows: in aPriori, select a GCD in the **Geometric Cost Drivers** pane. The `gcd` object field names are the same as the names displayed in the **Name** column of the **Geometric Cost Drivers** pane, modified to remove spaces and to make the initial character lower case. Note that for the name, `attribute-name`, of any GCD attribute, the value of `gcd.<attribute-name>` is the same as the value of `op.<attribute-name>`.

global

The value of this identifier is an object that is used to pass information among different CSL modules. You can add a field to `global` by using a formula (within a set block) whose left hand side is a complex identifier consisting of “`global`” followed by dot followed a new field name. See [Set Blocks](#) for information on set blocks.

Use this identifier to pass information from one process or operation to a sibling process or operation in the current routing or operation sequence, during process or operation taxonomy evaluation. It is set to null after each taxonomy evaluation pass (including after the yield pass).

HoleType

The value of this identifier is an enumeration object with the following fields:

- BLIND
- THROUGH
- OBSCURED
- UNKNOWN

InclusionStatus

The value of this identifier is an enumeration object with the following fields:

- AUTO_INCLUDE
- USER_INCLUDE

One of these is the value of the `inclusionStatus` property of any active node in the current process routing or operation sequence. The value indicates whether the node has been manually or automatically included in the routing or sequence. If the node is not active in the routing (for example, if it is optional and infeasible), the value of the `inclusionStatus` property is `null`.

KeywayBottomType

The value of this identifier is an enumeration object with the following fields:

- FLAT
- ARC

- BATHTUB
- UNKNOWN

KeywayEndType

The value of this identifier is an enumeration object with the following fields:

- THROUGH
- OPEN
- CLOSED
- UNKNOWN

material

The value of this identifier is an object that has one field for each attribute of the current material. You can view material attributes for the current process group in the **Material Composition** table of the **Material Selection** dialog (click the **Material** button in aPriori's **Manufacturing Process** pane) .

machine

The value of this identifier is an object that has one field for each attribute of the current machine. You can view machine attributes for a given process in the **Edit Machine Selection** dialog (right click a process name in aPriori's **Manufacturing Process** pane, and select **Machine Selection**). Note that this is in scope only if a machine has been selected for the current process.

machines

The value of this identifier is a collection that has one element for each machine available for the current process. You can view available machines for a given process in the **Edit Machine Selection** dialog (right click a process name in aPriori's **Manufacturing Process** pane, and select **Machine Selection**). Note that this is in scope only for machine selection modules.

Lookup table identifiers

There is one identifier for each lookup table associated with current process group, and , if the current node is a process node, each lookup table associated with the current node and all its ancestors up to and including the current process. Each identifier is the short name (*not* pathname) of a table. The value of each identifier is a table accessible through the use of a query expression. all the lookup tables associated with the process group, the current node and all its parents in the process-tree hierarchy. (So, if the current node is an operation, the parent process's lookup tables won't be available, but parent nodes in the operation sequence will be.)

op

This is an object that has one field for each already-evaluated output formula of a module associated with the current routing node, as well as one field for each attribute of the current GCD. The field names and values are the same as the formula and attribute names and values.

You can view a list of these formulas and attributes as follows:

- Formula names:

- In CMWB, select the operation or process corresponding to the current routing node and click the **CSL Modules** tab.
- See Outputs for information on standard, implicit output formulas for each CSL module type.
- **GCD attributes:** in aPriori, select a GCD in the **Geometric Cost Drivers** pane. The `gcd` object field names are the same as the names displayed in the **Name** column of the **Geometric Cost Drivers** pane, modified to remove spaces and to make the initial character lower case.

Other fields of `op` include the following:

- `childOps`: collection with an element for each already-costed child operation of the child operation. This allows recursive access to descendants of the current process or operation.
- `displayName`: name of the operation as translated into the local language.
- `ignoreFeasibility`: `true` if the current routing node was explicitly specified by the user to be included in the current routing regardless of feasibility; `false` otherwise.
- `isManualOverride`: `true` if the current routing node was explicitly selected by the user to be included in the current routing; the value is `false` otherwise.
- `inclusionStatus`: indicates whether the current node is user-included or auto-included in the current process routing or operation sequence. See `InclusionStatus` for possible values of this field.

part

The value of this identifier is an object with one field for each attribute of the current part, including GCDs and production information.

This identifier is commonly used to access the current part's geometric properties, as in, for example, `part.height` and `part.volume`. A collection of the current part's GCDs is designated by `part.childArtificats`.

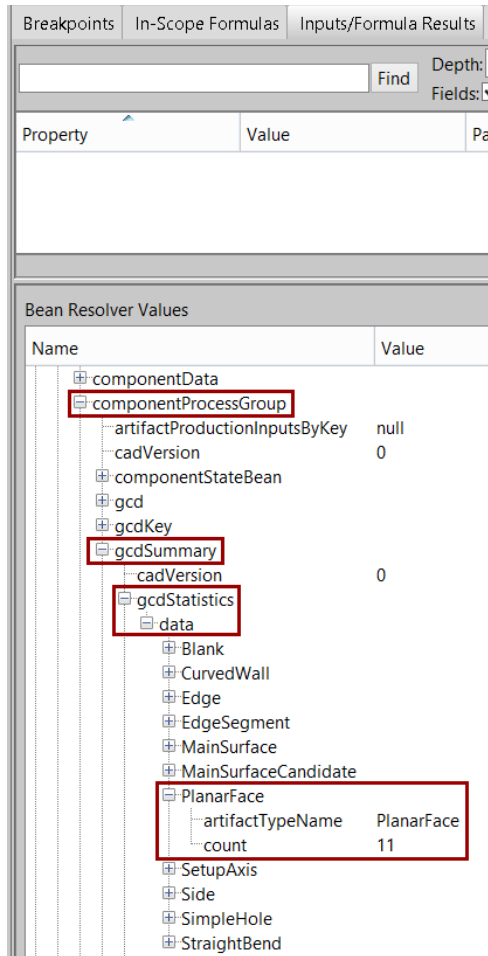
A property path of the following form allows access to the count of given *GCD-type* for the current part:

```
part.componentProcessGroup.gcdSummary.gcdStatistics.data.<GCD-type>.count
```

So, for example, the following expression can be used to retrieve the count of SimpleHoles for the current part:

```
safeEval(part.componentProcessGroup.gcdSummary.gcdStatistics.data.SimpleHole.count, 0)
```

The structure of the property path can be seen in the **Inputs/Formula Results** tab of the Cost Model Workbench Debugger:



The `part` object also has a field, `customAttribute`, which designates an object that has a field for each user-defined attribute (UDA). So the form for accessing UDAs in CSL is `part.customAttributes.<UDA-name>`

where UDA-name is the name that appears in the **Name** column of the System Admin UDA table.

PartingType

The value of this identifier is an enumeration object with the following fields:

- FLAT
- NON-FLAT
- UNKNOWN

passNumber

The value of this identifier is the current pass number, 1, 2, 3, or 4, if the current node is an operation node or operation sequence branch node. See Cost Engine Details for more information on pass numbers.

perCosting

The value of this identifier is an object that is used to pass information among different CSL modules. You can add a field to `perCosting` by using a formula (within a set block)

whose left hand side is a complex identifier consisting of “perCosting” followed by dot followed a new field name. See Set Blocks for information on set blocks.

This identifier is set to null after every complete costing. Use this object for expensive computations that won't vary by routing. *Caution:* in some cases, the GCD hierarchy itself can vary by routing (for example, when aPriori creates RingedHoles for turning routings).

perRouting

The value of this identifier is an object that is used to pass information among different CSL modules. You can add a field to perRouting by using a formula (within a set block) whose left hand side is a complex identifier consisting of “perRouting” followed by dot followed a new field name. See Set Blocks for information on set blocks.

This identifier is set to null after each routing evaluation. Note that there are potentially multiple evaluations of each routing—one for each stock.

plant

This is an object that has one field for each plant variable of the current VPE and process group. The field name is the same as the name of the corresponding variable; the field value is the value of the corresponding variable. You can view a list of these variables (and their values) as follows:

- Global plant variables: In VPE Manager, open the current VPE and select the top-level node in the navigation pane. The plant object field names are the same as the names displayed in the VPE Manager, modified to remove spaces and to make the initial character lower case.
- Process-group-specific plant variables: In CMWB, select **Global Cost Model Information > Cost Model Variables**.

RelationType

The value of this identifier is an enumeration object with the following fields:

- LIES_NEAR
- LIES_OUTSIDE
- INTERSECTS
- PARALLEL
- ENDS_ON
- COAXIAL
- ADJACENT
- LIES_ON
- IS_ABOUT
- IS_ORTHOGONAL
- IS_ACCESSIBLE_FROM

results

This is a collection with one object for each number-valued output identifier associated with the current process or operation or any of its child processes or operations. If there

are multiple children that define an output with the same name, the corresponding field of `results` is the sum of the output values. All currency values are converted to the current VPE's currency. `results` is in scope for taxonomy modules only.

setup

The value of this identifier is an object with one field for each process setup option associated with the current process. The field names are the option names. You can view the process setup options for a given process as follows:

- Double click the process under **Processes, GCDs & Operations** in the CMWB navigation pane.
- Click the **Process Setup Options** tab.

SetupDirectionType

The value of this identifier is an enumeration object with the following fields:

- NORMAL
- PARALLEL
- OBLIQUE
- PARALLEL_OBSTRUCTED
- NORMAL_OBSTRUCTED
- OBLIQUE_OBSTRUCTED
- UNKNOWN

SetupType

The value of this identifier is an enumeration object with the following fields:

- PRIMARY
- SECONDARY
- PRIMARY_ROTATIONAL
- SECONDARY_ROTATIONAL
- OTHER

site

The value of this identifier is an object that has one field for each site variable of the current process group. The field names and values are the same as the variable names and values. You can view a list of these attributes (and their values) by selecting **Process Group Site Variables** from the aPriori VPE Toolset.

stock

The value of this identifier is an object with one field for each attribute of the current material stock. Note: this is only in scope for modules associated with primary processes and operations; it is not available for secondary processes and operations. Note that this is in scope only if the stock selection module has been evaluated for the current part.

tool

The value of this identifier is an object that represents the tool chosen for the current node. Note that this is in scope only for modules associated with an operation, and is non-null only if a tool has been selected for the current operation.

tools

The value of this identifier is a collection with one element for each tool available for the current operation. Note that this is in scope only for tool selection modules.

toolShop

The value of this identifier is an object with one field for each variable of the toolshop that is associated directly with either the current node, one of the current node's parent nodes, or one of the current node's descendant nodes (if exactly one descendant node has a tool shop).

TurningApproach

The value of this identifier is an enumeration object with the following fields:

- OD
- FACE_A
- FACE_B
- EITHER_FACE
- ID
- SLANTED
- NONE

VoidShape

The value of this identifier is an enumeration object with the following fields:

- EXTRUDED
- COMPLEX
- UNKNOWN

Outputs

The cost engine, by evaluating a CSL module, establishes values for all the CSL module's associated output identifiers. The output values established in the course of costing a part become available to other CSL modules during the costing process, as well as to the aPriori GUI when costing is complete.

What counts as an output identifier for a given module depends on the module's type. CSL supports the following module types:

- Process taxonomy
- Operation taxonomy
- Process feasibility (routing rules)
- Operation feasibility (routing rules)

- Material stock selection
- Machine selection
- Tool selection
- Template pruning
- Process optionality
- Operation optionality
- Operation Precedence
- Operation Type Precedence
- Operation compatibility
- Zero or More Operation
- Idle Station Count
- Form merge rules
- Bend merge rules
- Shear form merge rules

An output formula is a formula whose left hand side is an output identifier. It specifies the value of one output (by assigning a value to the identifier). If and when the formula is evaluated, the assignment is made. See Formulas for information on formula evaluation.

Taxonomy Modules

Process and operation taxonomy modules include the following output identifiers:

- Formula names listed in the module's formula table on the **CSL Modules** tab of the module's associated process, operation, or template node in the CMWB.
- Formula names listed in the **costTaxonomy** formula table on the **CSL Modules** tab of the module's associated process group (select **Globally Available CSL/Lookup Tables**).

Feasibility Modules

Process and operation feasibility modules, in addition to returning a boolean, include as output identifiers all formula names listed in the module's formula table on the **CSL Modules** tab of the module's associated process, operation, or template node in the CMWB. By default there are none, but users can specify custom outputs.

Stock Selection Modules

Stock selection modules include the following output identifiers:

- `validMaterialStocks`
- `sortedMaterialStocks`
- `bestMaterialStock`
- Formula names listed in the module's formula table on the **CSL Modules** tab of the module's associated process node in the CMWB. By default there are none, but users can specify custom outputs.

Stock selection modules can also have *contingent* output formulas, that is, formulas that are treated as output formulas when a related formula evaluates to a specified boundary value.

For the machining process group, the following are treated as output formulas when `validMaterialStocks` evaluates to null or an empty collection:

- `virtualStockLength`
- `virtualStockWidth`

For the sheet metal process group, the following are treated as output formulas when `validMaterialStocks` evaluates to null or an empty collection:

- `virtualStockLength`
- `virtualStockWidth`
- `virtualStockHeight`
- `virtualStockThickness`
- `virtualStockInsideDiameter`
- `virtualStockOuterDiameter`

Machine Selection Modules

Machine selection modules include the following output identifiers:

- `machine`
- Formula names listed in the module's formula table on the **CSL Modules** tab of the module's associated process node in the CMWB. By default there are none, but users can specify custom outputs.

Tool Selection Modules

Tool selection modules always include `tool` as an output identifier.

Tool selection modules have as *contingent* output formulas the formulas listed in the module's formula table on the **CSL Modules** tab of the module's associated process, operation, or template node in the CMWB (by default there are none, but users can specify custom outputs). When `tool` evaluates to a non-null value, the formula names in this table are treated as output identifiers.

Template Pruning Modules

Template pruning modules have no associated output formulas; they simply return a boolean.

Optionality Modules

Optionality modules have no associated output formulas; they simply return a boolean.

Return Values

The code in a CSL module specifies a return value by using rules. Each rule, upon evaluation, either succeeds or fails. If a rule fails, rule evaluation terminates and the value **false** is returned. When all the rules in a module succeed, the value **true** is returned. The returned value is used to help determine process and operation routing—see

Working with Cost Model Logic for more information on how return values help determine routing.

Only the following types of CSL modules return a value:

- Process feasibility
- Operation feasibility
- Template pruning
- Process optionality
- Operation optionality
- Operation Precedence
- Operation Type Precedence
- Operation compatibility
- Form merge rules
- Bend merge rules
- Shear form merge rules

Imports

An import directive consists of the import keyword followed by a file name. When the cost engine processes an import directive, it effectively includes the text of the specified file in the current CSL module, for the purposes of the current module evaluation.

When the cost engine evaluates a CSL module, it first performs imports by doing the following:

- Importing the `constants.csl` for the current process group, if it exists. You can access this file as follows: Double click **Globally Available CSL/Lookup Tables** under **Global Cost Model Information** in the navigation pane of the CMWB, select the **CSL Modules** tab, and click the **constants** folder.
- Importing the file specified by each import directive in the current module.

For taxonomy modules, the cost engine also imports formula names listed in the **costTaxonomy** formula table on the **CSL Modules** tab of the module's associated process group (click **Globally Available CSL/Lookup Tables**).

Each import statement has the following form:

```
import <simple-identifier>.csl
```

`simple-identifier` is the short name (not pathname) of the file to be imported. Each import directive ends with a line break.

The file must be declared as a library file in the cost model. To view declared library files for a given cost model, **Globally Available CSL/Lookup Tables** under **Global Cost Model Information** in the navigation pane of the CMWB, and select the **Library CSL** tab.

Formulas and Rules

The primary syntactic constructs in CSL are *formulas* and *rules*, and *advice rules*.

CSL supports parameterized formulas in the form of *function definitions*. Formulas can also be enclosed in *set blocks*, to ensure their evaluation even if they are not output formulas.

In addition, each rule can have an associated *message* and associated *advice* which are evaluated if the rule returns false. Each advice rule can also have associated advice, which is evaluated if the advice rule returns true.

This section covers the following constructs:

- Formulas
- Set Blocks
- Rules
- Messages
- Advice
- Advice Rules
- Function Definitions

Formulas

Each formula has the following form:

```
identifier = expression
```

A formula consists of an identifier (the *formula name*, also referred to as the *left-hand side*) followed by “=” followed by an expression (referred to as the *formula’s expression* or the *right hand side*). Each formula ends with a line break.

When the cost engine evaluates a formula, it does the following:

- Evaluates the formula’s expression. (See Expressions for information on expression evaluation.)
- Assigns the result of the expression evaluation to the formula’s identifier.

The cost engine, in the course of evaluating a formula’s expression, sometimes evaluates other formulas. In particular, if the expression contains an identifier that is not one of the current module’s input identifiers, the cost engine determines the identifier’s value by evaluating the rule or formula (in the current module) that has this identifier as its name.

When the cost engine evaluates a module, it evaluates the following formulas:

- All the module’s output formulas, that is, all formulas whose left-hand side is an output identifier
- All formulas in the module’s `set` blocks
- All the module’s formulas whose left-hand side appears as an identifier in an expression that the cost engine evaluates.

The cost engine evaluates *only* these formulas. Some formulas in a module might not be evaluated.

(The cost engine caches the results of formula evaluation, so that it typically evaluates each formula at most once during evaluation of a given module.)

See the following sections for information on each syntactic category mentioned above:

- Identifiers and Literals
- Expressions
- Set Blocks

Set Blocks

Each set block has the following form:

```
set {  
    <formula> [  
    <formula>]*  
}
```

A set block consists of the keyword “set” followed by a sequence of formulas enclosed in braces.

A formula that is *not* within a set block is evaluated only if either it is an output formula or its evaluation is required (directly or indirectly) for the evaluation of an output formula (see Formulas and Outputs). A formula that *is* within a set block is always evaluated as part of the evaluation of the module that contains it. The formulas in a set block are evaluated only after evaluation of all output formulas that are not contained in a set block.

Set blocks are typically used to pass information to other modules by setting fields of input objects. The following input objects are particularly useful for this purpose:

- **global**: use this identifier to pass information from one process or operation to a sibling process or operation in the current routing or operation sequence, during process or operation taxonomy evaluation. It is set to null after each taxonomy evaluation pass (including after the yield pass).
- **perCosting**: this identifier is set to null after every complete costing. Use this object for expensive computations that won't vary by routing. *Caution*: in some cases, the GCD hierarchy itself can vary by routing (for example, when aPriori creates RingedHoles for turning routings).
- **perRouting**: This identifier is set to null after each routing evaluation. Note that there are potentially multiple evaluations of each routing—one for each stock.

Each of these inputs allows you to introduce new fields. For example, you can introduce new fields of `global` by using a formula whose left hand side is a complex identifier consisting of “`global`” followed by dot followed a new field name. See Inputs for information on input objects.

Example

```
set {  
    global.numScrapPartsDownStream = {
```

```
    numScrapParts if safeEval(global.numScrapPartsDownStream, null)
    == null
    numScrapParts + global.numScrapPartsDownStream otherwise
  }
}
```

Example

```
set {
  global.foo = 5
  part.bar = 'baz'
}
```

Rules

A rule has the following form:

```
{<boolean-expression> | Rule <simple-identifier> : <boolean-expression>}
```

Each rule is either named or unnamed. A named rule consists of a simple identifier (the rule's name) and a boolean expression. An unnamed rule consists only of a boolean expression. Each rule ends with a line break.

When the cost engine evaluates a rule, it evaluates the boolean expression (see Boolean Expressions for information on boolean expression evaluation). For a named rule, the cost engine assigns the value (true or false) of the boolean expression to the rule's name.

The cost engine, in the course of evaluating a rule's conditional expression, sometimes evaluates formulas. In particular, if the expression contains an identifier that is not one of the current module's input identifiers, the cost engine determines the identifier's value by evaluating the formula (in the current module) that has this identifier as its left-hand side.

There are two possible outcomes of the evaluation:

- The conditional expression evaluates to `false`. In this case, the rule *fails*. When a named rule fails, the cost engine does all the following:
 - Evaluates the message construct (if there is one) whose name matches the rule's name (see [Messages](#)).
 - Evaluates the advice construct (if there is one) whose name matches the rule's name (see [Advice](#)).
 - Terminates evaluation of the current module's rules, and returns `false` as the module evaluation result.
- The conditional expression evaluates to `true`. In this case, the rule *succeeds*, and module evaluation continues. If all the rules in a module evaluate to `true`, module evaluation returns `true`.

A module's returned value is used to help determine process and operation routing—see Working with Cost Model Logic for more information on how return values help determine routing.

See the following sections for information on each syntactic category mentioned above:

- Boolean Expressions

- Simple Identifiers
- Messages

Messages

A message construct has the following form:

```
Message <simple-identifier> : <string-expression>
```

Each message statement consists of an identifier (the message name) and a string expression. Each message ends with a line break. The cost engine executes a message statement only when a rule with a matching name fails. See [Rules](#) for information on rules.

When the cost engine evaluates a message statement, it evaluates the string expression, and appends the result to the cost engine log. See [String Expressions](#) for information on string expression evaluation.

If a rule has more than one matching message, an error occurs when the module is saved.

See the following sections for information on each syntactic category mentioned above:

- String Expressions
- Simple Identifiers
- Rules

Advice Rules

An advice rule has the following form:

```
AdviceRule <simple-identifier> : <boolean-expression>
```

Each advice rule consists of a simple identifier (the advice rule's name) and a boolean expression. Each advice rule ends with a line break.

When the cost engine evaluates an advice rule, it evaluates the boolean expression. When the conditional expression evaluates to **true**, the cost engine evaluates the advice construct whose name matches the advice rule's name (see [Advice](#)).

See the following sections for information on each syntactic category mentioned above:

- Boolean Expressions
- Simple Identifiers
- Advice

Advice

An advice construct has the following form:

```
Advice <simple-identifier> : <function-invocation>
```

Each advice construct consists of an identifier (the advice name) and a call to the function **dtcMessage** (see [Miscellaneous Functions](#)). Each advice construct ends with a line break, and line breaks are allowed between parameters of the function call. The cost engine executes an advice construct only when at least one of the following is true:

- Some *rule* with a matching name evaluates to **false**.
- Some *advice rule* with a matching name evaluates to **true**.

When the cost engine evaluates an advice construct, it evaluates the function invocation, and appends the result, a **dtcMessage** *object*, to the list of **dtcMessage** objects for the current process routing. **dtcMessage** objects contain information about the context in which the associated rule failed or in which the associated advice rule succeeded; this information is used to populate the Design to Cost panels in the aPriori Professional interface (see Design to Cost in the aPriori Professional *User Guide*). For more information on **DtcMessage** objects, see **dtcMessage** in [Miscellaneous Functions](#).

If a rule or advice rule has more than one matching advice construct, an error occurs when the module is saved.

See the following sections for information on each syntactic category mentioned above:

- [Simple Identifiers](#)
- [Function Invocations](#)
- [Rules](#)
- [Advice Rules](#)

Function Definitions

Each function definition has the following form:

```
<simple-identifier>(<simple-identifier>[, <simple-identifier>]*) =  
<expression>
```

A function definition can be thought of as a parameterized formula. Syntactically, it consists of two parts separated by “=”:

- Left hand side: the function name followed by a parentheses-enclosed, comma-separated list of formal parameters. The function name and formal parameters are all simple identifiers.
- Right hand side: an expression that may contain occurrences of the formal parameters specified in the left hand side

Each function definition ends with a line break.

The cost engine evaluates a function invocation by evaluating the right hand side of the function definition (in the same module) with a matching name, substituting the invocation’s actual parameters for the definition’s corresponding formal parameters. See [Function Invocations](#) for information on function invocations.

Note that CSL provides a number of predefined functions. See [Predefined Functions](#) for more information.

See the following sections for information on each syntactic category mentioned above:

- [Simple Identifiers](#)
- [Expressions](#)
- [Function Invocations](#)

Expressions

An `expression` in CSL has one of the following forms:

- `arithmetic-expression`. See Arithmetic Expressions.
- `string-expression`. See String Expressions.
- `boolean-expression`. See Boolean Expressions.
- `conditional-expression`. See Conditional Expressions.
- `function-invocation`. See Function Invocations.
- `query-expression`. See Query Expressions.
- `foreach-expression`. See Foreach Expressions.

Arithmetic Expressions

An arithmetic expression designates a floating point number. It has the following form:

`<term> [<binary-operator> <term>]`

A `binary-operator` is one of the following:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `^` (exponentiation)

A `term` is one of the following:

- Numeric literal. See Numerical Literals.
- Identifier (must be number-valued). See Identifiers and Literals.
- Function invocation (must be number-valued). See Function Invocations.
- Arithmetic Expression. See Arithmetic Expressions.

Arithmetic expressions can be grouped using parentheses.

Conventional arithmetic precedence rules are followed:

- Parentheses
- Function Invocation
- Unary operators
- Exponentiation (symbol `^`)
- Multiplication/division (symbols `*` and `/`)
- Addition/subtraction (symbols `+` and `-`)
- Left to right evaluation (textual order)

String Expressions

A string expression designates a string, that is, a sequence of characters. It has the following form:

```
<string-literal>
```

```
<identifier>
```

Here the `identifier` must be string-valued.

In the message clause of a rule, the operator `+` can be used to form a complex string expression:

```
<string-expression> + <string-expression>
```

The result of evaluation of a complex string expression is the result of concatenating the values of the `string-expressions`.

See Identifiers and Literals for information on string literals and identifiers.

Boolean Expressions

A `boolean-expression` designates `true` or `false`. CSL boolean expressions use conventional infix syntax:

```
[~]<term> [<binary-logical-operator> [~]<term>]*
```

A `binary-logical-operator` is one of the following:

- `and`
- `or`

Each `term` is one of the following:

- Boolean literal. See Boolean Literals.
- Identifier (must be boolean valued). See Identifiers and Literals.
- Function invocation (must be boolean valued). See Function Invocations.
- Arithmetic comparison. See below.
- Boolean expression. See Boolean Expressions.

An `arithmetic-comparison` has the following form:

```
<arithmetic-expression> <comparison-operator> <arithmetic-expression>
```

A `comparison-operator` is one of the following:

- `>` (greater than)
- `>=` (greater than or equal to)
- `<` (less than)
- `<=` (less than or equal to)
- `==` (equal to)
- `<>` (not equal to)

Boolean expressions can be grouped using parentheses.

Conventional precedence rules for conditional expressions are as follows:

- Parentheses (explicit expression nesting)
- ~ (logical negation)
- Arithmetic comparison
- and
- or
- Left to right (textual order)

Conditional Expressions

A conditional-expression has one of one of the following forms:

```
{ <expression> if <boolean-expression> [
  <expression> if <boolean-expression>]*
  <expression> otherwise }
```

```
{ if (<boolean-expression>) {[with-clause] <expression>} [
  else if (<boolean-expression>) {[with-clause] <expression>}]*
  else {[with-clause] <expression>} }
```

Here `expression` must *not* be a conditional expression; nested conditional expressions are not allowed. See Expressions for information on other kinds of expressions.

Conditional expressions of the first form can contain a line break after a `boolean-expression`. Conditional expressions of the second form can contain a line break anywhere except within conjunctions or disjunctions (that is, except in between `and`/`or` clauses in a `boolean-expression`)

Each pair of curly braces within a conditional of the second form is optional in the absence of a `with-clause`.

A conditional expression evaluates to the value of the `expression` associated with the first `boolean-expression` that evaluates to true. If no `boolean-expression` evaluates to true, the conditional expression evaluates to the value of the `expression` associated with the `otherwise` or `else`.

Function Invocations

Each function invocation has the following form:

```
<simple-identifier>(<actual-parameter>[, <actual-parameter>]*)
```

A function invocation consists of a function name followed by a parentheses-enclosed, comma-separated list of actual parameters.

Each `actual-parameter` is one of the following:

- `arithmetic-expression` (see Arithmetic Expressions)
- `boolean-expression` (see Boolean Expressions)
- `string-expression` (see String Expressions).

The cost engine evaluates a function invocation by evaluating the right hand side of the function definition with a matching name, substituting the invocation's actual parameters for the definition's corresponding formal parameters. See Function Definitions for information on function definitions.

The i^{th} formal parameter in the left hand side of a function definition corresponds to the i^{th} actual parameter in the function invocation (for any positive integer, i).

The cost engine looks for a function definition with a matching name as follows:

- First, the following predefined functions are considered:
 - `is<ArtifactTypeName>()`
 - `getPrevToolRow()`
 - `getNextToolRow()`
 - `getPrevMillRow()`
 - `getNextMillRow()`
 - `snapDownLookUp()`
 - `getPrevBestDiamTolRow()`
 - `getNextBestDiamTolRow()`
- Next, the cost engine considers functions defined in the current module (or in a file imported by the current module).
- Finally, the cost engine considers other predefined functions.

Note that this means that user-defined functions override predefined functions with the same name, except for the predefined functions listed above; the functions listed above cannot be overridden.

Note that when a function invocation returns an object, it can be followed by object field access dot notation, as in the following:

```
myFunction(5).id
```

Such a construct is one form of `complex-identifier`. See Identifiers and Literals for more information on identifiers.

Note that CSL provides a number of predefined functions. See Predefined Functions for more information.

Query Expressions

Query expressions are used to retrieve and aggregate values from collections. Their semantics is essentially the same as SQL query semantics. A query expression evaluates to a collection, object, string, number, or boolean value.

Each query expression has the following form:

```
select [distinct] <select-specification>
from <identifier> [ [as] <simple-identifier>]
[where <boolean-expr>]
[group by <arithmetic-expr>[, <arithmetic-expr>]*]
[order by <arithmetic-expr> [asc | desc][, <arithmetic-expr> [asc | desc]]*]
```

Select clause

Each `select-specification` has the following form:

```
{<select-arg> | <simple-identifier>(<select-arg>)}
```

Here the `simple-identifier` names one of the following query aggregate functions:

- `first`: returns the first element of the intermediate query results.
- `last`: returns the last element of the intermediate query results.
- `sum`: returns the sum of the values in the intermediate query results.
- `min`: returns the smallest value in the intermediate query results.
- `max`: returns the largest value in the intermediate query results.
- `count`: returns the number of elements in the intermediate query results.

Each `select-arg` is one of the following:

- `*`
- `<complex-identifier>`

Note that `select-arg` can be the name of a formula whose right hand side includes occurrences of the `from` clause's `simple-identifier`. See `From clause` for information on the `from` clause.

From clause

The `identifier` in the `select` clause must evaluate to the collection to be queried. Note that tables in CSL are represented as collections whose elements are objects with one field for each table column.

The `simple-identifier` in the `select` clause is an alias for a single element in the collection being queried. It can occur in the following components of the query expression:

- `select-specification` in the `select` clause
- `boolean-expression` in the `where` clause
- `arithmetic-expressions` in the `group by` clause
- `arithmetic-expressions` in the `order by` clause

This alias can also occur in the right hand side of formulas whose evaluation is necessitated by evaluation of expressions in the query. The `simple-identifier` is has file scope.

Examples

```
numManualMigWelds = _
  select count(w) from welds w _
  where w.type != 'Spot' and not w.isRobotic
```

```
componentMass = _
  select sum(c.weight) from part.subcomponents c
```

```
finishMass = _
```

```

select sum(w.weldWeight) from childOps w _
where isWeld(w)

weldWeight = _
select sum(op.weldWeight) from childOps op

numCostableWelds = _
select count(a) from part.childArtifacts a _
where isWeld(a) and a.costable

computedClampTime = _
select first(entry.loadTime) from xtraLoadTime entry _
where entry.weight >=gcd.weight
order by entry.loadTime // time (secs) for load part

coreboxCostVoids =
select sum(operation.hardToolingCost) from childOps operation _
where operation.artifactType.name =='Void' _
group by _
    adaptiveRound(operation.volume, _
        voxelVolumeError(voidCoreArea(operation))), _
    adaptiveRound(operation.boxLength, voxelLengthError), _
    adaptiveRound(operation.boxWidth, voxelLengthError), _
    adaptiveRound(operation.boxHeight, voxelLengthError)

```

Foreach Expressions

Foreach expressions provide collection access. Simple foreach expressions have the following form:

```

foreach (<identifier-1> : <expression-1>) <identifier-2>(<identifier-3>) {
    <identifier-3> = <expression-2>
}

```

Here is an example:

```

x = foreach (tool : tools) getFirst(t) { _
    t = { tool if tool.diameter > 5 null otherwise }
}

```

Such a foreach expression specifies:

- Iteration variable (*identifier-1*, which must be a simple identifier)
- Collection (*expression-1*, which must be collection-valued)
- Reduction function (*identifier-2*, which must name a function)
- Formula containing the iteration variable (named by *identifier-3*)

Evaluation of a foreach expression effectively does the following:

- Iterates through the collection elements one at a time, binding each element to the iteration variable in turn. For each iteration, it does this:
 - Evaluates the formula, with the current element bound to the iteration variable within the formula.

- Adds the result of the formula evaluation to a *results collection*.
- Applies the reduction function to the results collection, once all elements have been processed, yielding the foreach expression's final result.

The cost engine uses knowledge of the semantics of the reduction functions to optimize evaluation of foreach expressions, so it does not always actually process every collection element.

CSL supports the following reduction functions:

- **getFirst**: Returns the first non-null element of the results collection.
- **getAll**: Returns the entire results collection.
- **getAllFlatten**: Returns a flattened results collection, that is, returns a collection containing all non-collection elements in the results collection, together with the non-collection elements of any collection elements of the results collection, and so on, recursively.
- **getMax**: Returns the element of the results collection that has the maximum numeric value.
- **getMin**: Returns the element of the results collection that has the minimum numeric value.
- **getSum** : Returns the sum of the elements of the results collection.
- **getProduct**: Returns the product of the elements of the results collection.
- **getStats**: Returns an object that contains one field for each of a variety of statistical properties of the results collection (including mean, standard deviation, minimum, maximum, and count--see <http://commons.apache.org/math/apidocs/org/apache/commons/math3/stat/descriptive/DescriptiveStatistics.html>)
- **getObjectWithMax**: Returns the value of the iteration variable (an element of the collection being iterated over) associated with the maximum element of the results collection. Consider a foreach expression that evaluates a `cycleTime` formula for each operation in a collection. While `getMax(cycleTime)` returns a number, `getObjectWithMax(cycleTime)` returns the operation with the maximum cycle time.
- **getObjectWithMin**: Returns the value of the iteration variable (an element of the collection being iterated over) associated with the minimum element of the results collection. Similar to `getObjectWithMax`.
- **buildGroup**
- **getAllDistinct**: Returns the results collection with duplicates removed.

Examples:

```
x = foreach (tool : select * from tools where diameter > 5) getFirst(t)
{
  _
  t = tool.diameter
}
```

```
x = foreach (tool : select * from tools where diameter > 5) getFirst(t)
{
  _
```

```
t = foreach (t : tool.machines) getAll(q) { q = .... }
}
```

With Clause

Foreach and some if expressions can include auxiliary formulas whose evaluation is required for evaluation of the primary (output) formula:

```
foreach (<identifier-1> : <expression-1>) <identifier-2>(<identifier-3>) {
  with {
    formula [
      formula]*
  }
  <identifier-3> = <expression-2>
}
```

Here is an example:

```
foreach ( t : tools)  getAll(f) { _
  with { _
    z = t.diameter *2
    q = select * from t.machines....
    r = z * 10
  }
  f = { t if (r > 5 and q != null) null otherwise }
}
```

Like Expressions

Like expressions can be used to determine if a given string matches a given regular expression. Each `like` expression has the following form:

```
{<string-expression> | <function-invocation>} like
{<string-expression> | <function-invocation>}
```

If an operand is a function invocation, it must be string-valued.

The right operand must evaluate to a regular expression, as described in <http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>, with the addition that the following characters (supported by SQL `like`) are allowed:

- `%` : matches an arbitrary sequence of characters (equivalent to `.*` in Java's regular expression dialect)
- `_` : matches a single arbitrary character (equivalent to `.` in Java's regular expression dialect)

A `like` expression evaluates to true if the value of the left operand matches the regular expression pattern designated by the right operand. It evaluates to false otherwise.

Example

```
isCore(op) = (op.name like '%Coring')
```

Identifiers and Literals

An `identifier` in CSL has one of the following forms:

- `simple-identifier`. See Simple Identifiers.
- `complex-identifier`. See Complex Identifiers.

Simple Identifiers

A legal simple identifier consists of a letter (A-Z, a-z) or underscore (`_`) followed by any number of letters, numbers or underscores. Note that dashes (`-`) are not legal in identifiers (as they are interpreted as minus signs). Also, unlike CSL Templates, identifiers in rules and formulas cannot include spaces.

Complex Identifiers

A complex identifier can have either of the following forms:

```
<simple-identifier>.<simple-identifier>[.<simple-identifier>]*
```

```
<function-invocation>.<simple-identifier>[.<simple-identifier>]*
```

See Simple Identifiers for information on simple identifiers. See Function Invocations for information on function invocations.

The function invocation or simple identifier to the left of the dot must be object valued.

A complex identifier can also have either of the following forms:

```
<simple-identifier>[string-literal]
```

```
<simple-identifier>[numerical-literal]
```

Note that the square brackets in the two lines above do *not* indicate optionality—they are not used as meta-characters, here; square brackets characters occur as part of these types of complex identifiers.

See Numerical Literals for information on numerical literals. See String Literals for information on string literals.

In these cases, the simple identifier to the left of the dot must be map valued. See also Map Functions.

Examples

```
part.totalCost  
myFunction(5).id  
myMap[3]  
myMap[ 'blue' ]
```

Numerical Literals

There are three forms of numeric literals: integer, real and scientific.

- Integer literals can begin with an optional sign (+,-) and consist of a sequence of digits (0-9)

- Real literals can begin with an optional sign (+,-) followed by a sequence of digits (0-9) followed by a dot (.) followed by a sequence of digits (0-9). Note that if there is a dot there must be one digit to the right of it. Thus "10." is not legal syntax, whereas "10.0" is.
- Scientific literals follow the same rules as real literals with the addition of an 'e' or 'E' followed by an optional sign followed by a sequence of (one or more) digits.

String Literals

Strings are enclosed by single quotes, ' , and can contain any letter or digit except single quote, backslash, \n or \r. These characters (or character sequences) have to be escaped with a backslash. So, a string literal consisting of a single backslash followed by a single quote would be '\\\''.

Boolean Literals

There are two Boolean literals: `true` and `false`.

Comments and Line Continuation

CSL supports three different forms for comments. Multiline comments are enclosed between `/*` and `*/`, while single line comments can follow either a double-slash `/**` or a pound sign `#`.

The following constructs must end with a line break:

- Import directive
- Formula
- Rule
- Message
- Function definition
- Set block

Line breaks within these constructs require use of the line continuation character (a space followed by an underscore--see below), except as follows:

- Conditional expressions can contain a line break after a boolean expression that follows `if`.
- Set blocks must contain a line break after each formula.
- Foreach expressions must contain a line break after each formula.

CSL permits VB-style line continuation:

```
// A space followed by the underscore is line continuation
a_bool = (part.thickness > machine.minThickness) && _
         (part.thickness < machine.maxThickness)
```

Predefined Functions

CSL supports the following categories of predefined functions:

- Numeric Functions
- String Functions
- List Functions
- Map Functions
- Node Attribute Functions
- Routing Navigation Functions
- Error Handling Functions
- Interpolation Function
- Miscellaneous Functions

Numeric Functions

CSL supports the following predefined numeric functions.

abs(x)

Returns the absolute value of x. The argument must be numeric.

acos(x)

Returns the arccosine of x in degrees.

asin(x)

Returns the arcsine of x in degrees.

atan(x)

Returns the arctangent of x in degrees.

cos(x)

Returns the cosine of the angle specified in degrees by x.

divZero(x, y)

Returns x/y if y \neq 0; returns 0 otherwise.

equalsEps(x, y, z)

Returns true if $\text{abs}(x - y)$ is strictly less than z; returns false otherwise

exp(x)

Returns the mathematical constant e raised to the x power.

interpolate(x, x0, x1, y0, y1)

Returns the y-coordinate of the point on the specified line whose x-coordinate is x . The specified line is the line containing both the points (x_0, y_0) and (x_1, y_1) . If x_0 equals x_1 , the function returns y_0 .

len(x)

Returns the cardinality of the collection x . An exception is thrown if x is not a collection.

ln(x)

Returns the natural logarithm of x .

max(x, y, z, ...)

Returns the maximum of the specified values.

maxNonNegativeNumber(x, y, z, ...)

Returns the maximum of the specified values, excluding negative numbers. If no argument is a non-negative number, `invalidScriptException` is thrown.

min(x, y, z, ...)

Returns the minimum of the specified values.

minNonNegativeNumber(x, y, z, ...)

Returns the minimum of the specified values, excluding negative numbers. If no argument is a non-negative number, `invalidScriptException` is thrown.

round(x)

Return the natural rounded value of (that is, the nearest integer to) x . As usual, for any integer, n , values in the interval $[n - 0.5, n + 0.5)$ round to n .

roundEps(x, y)

Return the value of rounding x , naturally, to the precision specified by y . Examples:

- `roundEps(1.45, 0.1)` returns 1.5
- `roundEps(1.44, 0.1)` returns 1.4

rounddown(x)

Return the greatest integer less than or equal to x (that is, returns the integer value of x , or floor of x).

roundup(x)

Returns the least integer greater than or equal to x (that is, returns ceiling of x).

sin(x)

Returns the sine of the angle specified in degrees by x .

sqrt(x)

Returns the positive square root of *x*.

sum(x, y, z, ...)

Returns the sum of arguments *y*, *z*, ... evaluated against all elements of the collection *x*.

tan(x)

Returns the tangent of the specified angle, *x*. The angle is specified in degrees.

getProperty(row, column)

Returns the value of the collection *row* for the attribute *column*.

String Functions

CSL supports the following predefined functions for the manipulation of strings.

Conversion Functions

These functions convert between strings and numbers.

asNumber(s)

Converts the string *s* to a double.

asString(n)

Converts the number *n* to a string. Note that *n* is interpreted as a double and may often have more digits after the decimal than expected. Use in conjunction with `mid()` to reduce to simple form if desired.

downCase(s)

Modifies a string so that it is all lowercase; that is, converts all uppercase alphabetic characters to their lowercase equivalent. If *s* is a number, this function converts it to a string.

htmlEscape(s)

Returns a version of the string with any reserved HTML characters replaced by their associated HTML entities. For example, '<' is replaced with '<'; '&' is replaced with '&'. The resulting string can be displayed safely in HTML-formatted fields.

upcase(s)

Modifies a string so that it is all uppercase; that is, converts all lowercase alphabetic characters to their uppercase equivalent. If *s* is a number, this function converts it to a string.

Index Functions

These functions return a string index that meets a specified condition.

index(s1, s2)

Returns the start index of the first occurrence of string s2 in string s1. The smallest index is 1 (that is, indexes are 1-based, not 0-based). Returns -1 if s2 does not occur in s1.

index(s1, s2, i)

Returns the start index of the first occurrence of string s2 that starts at or after index i in string s1. The smallest index is 1 (that is, indexes are 1-based, not 0-based). Note that `index(s1, s2, 1)` is equivalent to `index(s1, s2)`. Returns -1 if there is no such occurrence.

lastIndex(s1, s2)

Returns the start index of the last occurrence of string s2 in string s1. The smallest index is 1 (that is, indexes are 1-based, not 0-based). Returns -1 if s2 does not occur in s1.

lastIndex(s1, s2, i)

Returns the start index of the last occurrence of string s2 that starts at or before index i in string s1. The smallest index is 1 (that is, indexes are 1-based, not 0-based). Returns -1 if s2 does not occur in s1.

len(s)

Returns the length of the string s.

Substring Functions

These functions return a substring that meets a specified condition.

mid(s, i)

Returns the trailing substring of string s whose first character is at index i.

mid(s, i, j)

Returns the substring of string s that whose first character is at index i and whose last character is at index j-1. The character at index j is not included in the returned string.

prefix(s1, s2)

Returns the leading substring of string s1 whose last character is immediately before the first occurrence of s2 in s1. If s2 is not found or is equivalent to s1, the empty string is returned.

searchString(string, regular-expression)

Returns a list of sub-strings from the specified string that match the specified regular expression pattern. The second argument must evaluate to a regular expression, as described in <http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>. See also Like Expressions.

splitString(string, regular-expression)

Returns a list of sub-strings created by splitting the original string whenever the separator string is found. The separator string is typically a single character, but can be a regular expression. Any whitespace before or after each of the resulting sub-strings is automatically trimmed. Examples:

`splitString('abc:def:ghi', ':')` returns ['abc', 'def', 'ghi']

`splitString('abc.def.ghi', '\\.')` returns ['abc', 'def', 'ghi'] (Note that '.' has to be escaped as '\\' in the separator expression.)

suffix(s1, s2)

Returns the substring of `s1` that immediately follows the first character of the last occurrence of `s2` in `s1`. Typically `s2` is one character in length. Note that if `s2` is more than one character in length, the returned string includes all characters of `s2` except the first. If `s2` is not found or matches the last character of `s1`, the empty string is returned.

suffix2(s1, s2)

Returns the substring of string `s1` that immediately follows the last character of the last occurrence of `s2` in `s1`. If `s2` is not found, is equivalent to `s1`, or matches a trailing substring of `s1`, the empty string is returned.

List Functions

CSL supports the following functions for the manipulation of lists. The use of these functions is illustrated in an example, below.

asPair(object-1, object-2)

Creates a Pair containing the two objects. These objects can later be accessed using `pair.first` and `pair.second` respectively.

asList(element-1, ... , element-n)

Returns a new list containing the arguments as elements. If there are no arguments (as in `asList()`), an empty list is returned. (Note that an empty list is different from a list whose only member is null, which is returned by `asList(null)`.) The expression `listCons(a, asList())` returns the list containing `a`.

listCons(newHead, list)

Returns a new list that prepends `newHead` to `list`. (This is implemented without copying `list`; it is performed in constant time, that is, the amount of time it takes is the same regardless of the size of the list.)

listAsString(list, [separator])

Converts each element of the list (or any collection) into a string and concatenates them, separating them with the given separator. If unspecified, the default separator is ',' (comma-space), giving a similar result to `asString()`, but without the surrounding square brackets.

For example:

`listAsString(asList('one', 'two', 'three'))` - returns 'one, two, three'

`listAsString(asList('one', 'two', 'three'), '|')` - returns 'one | two | three'

`listAsString(asList(1, 2, 3))` - returns '1.0, 2.0, 3.0'

If `list` is null, an empty string is returned (the same as for an empty collection).

Otherwise, if `list` is not a collection the result matches `asString()`.

listFirst(list)

Returns the first element of `list`. (This is performed in constant time.)

listGet(list, i)

Returns *i*th element the list. List indexing is 1-based (that is, a list's first element has index 1 rather than index 0). If *i* is not a number between 1 and the size of the list, an exception is issued.

listRest(list)

Returns a list that includes everything in `list` except its first element. (This is performed in constant time; the list is not copied or modified.)

listReverse(list)

Returns a new list that contains the elements of `list` in reverse order. (The time to perform this increases linearly with the number of elements in the list.)

listSize(list)

Returns the number of elements in `list`. Returns 0 if `list` is not a list. *This function works if the underlying object is any type of Java Collection.*

Example

```
ListOfUpstreamSiblings(op) =
  ListOfUpstreamSiblings1(op,op.parent.children)

// if we run out of siblings, or we hit op, return the empty list
ListOfUpstreamSiblings1(op,siblings) = { _
  asList() _
  if (siblings == null or listFirst(siblings) == op)
  listCons(listFirst(siblings),ListOfUpstreamSiblings1(op,listRest(
  siblings)))_
  otherwise }
```

Note that recursion such as is used in this example does not scale with list size, and can lead to stack overflow errors with large lists.

Map Functions

CSL supports the following functions for the manipulation of maps (ordered or unordered collections of key-value pairs):

asMap(key-1, value-1, ... key-*n*, value-*n*)

Returns a new, unordered map containing the pair $\langle \text{key-}i, \text{value-}i \rangle$, for every *i* between 1 and *n*, inclusive. There must be an even number of arguments. The arguments must alternate between key and value, and must start with a key and end with a value. If there

are no arguments (as in `asMap()`), an empty map is returned. The map is implemented as a Java `HashMap`.

asOrderedMap(key-1, value-1, ... key-n, value-n)

Returns a new, ordered map containing the pair `<key-i, value-i>`, for every *i* between 1 and *n*, inclusive. There must be an even number of arguments. The arguments must alternate between key and value, and must start with a key and end with a value. If there are no arguments (as in `asMap()`), an empty map is returned. The map is implemented as a Java `TreeMap`.

mapGet(map, key)

Returns the value associated with the given key in the map. For Integer and String keys, you can also use the square-bracket notation, as in `myMap[3]` and `myMap['blue']`. Calls to `mapGet` work for any key type.

mapPut(map, key, value)

Adds a new key-value pair to an existing map.

Node Attribute Functions

CSL supports the following functions related to node attributes. The use of these functions is illustrated in Examples, below.

getAttributeValue(op, attributeName)

Returns, as a string, the value of the attribute named `attributeName` for the node `op`, if the attribute exists; returns `null` otherwise.

getAttributeValueAsBoolean(op, attributeName)

Returns, as a Boolean, the value of the attribute named `attributeName` for the node `op`, if the attribute exists; returns `false` otherwise.

getAttributeValueAsNumber(op, attributeName)

Returns, as a number, the value of the attribute named `attributeName` for the node `op`, if the attribute exists; returns `0.0` otherwise.

getNodeInTree(op, attributeName)

This function searches the process routing or operation sequence in which `op` occurs, as well as the ancestors of `op` in the process-operation hierarchy (see Hierarchies). The function returns a node with the specified attribute, if there is one, and returns `null` otherwise.

Note that the search includes optional nodes, regardless of whether they are active in the process routing or operation sequence. To determine if an optional node is active, check the value of its `inclusionStatus` property (see `InclusionStatus` and Examples).

Note that a process routing or operation sequence is a template instantiation (see Hierarchies), and so is potentially tree structured. The function searches all the nodes in that tree. The function also searches ancestors of `op` in the process-operation hierarchy;

so, for example, if `op` is a Bending operation on a SimpleHole child of a MultistepHole, the function checks the parent operation node, MultistepHolemaking, as well as all the nodes in the SimpleHole's operation sequence.

hasAttribute(op, attributeName)

Returns `true` if the node `op` has an attribute named `attributeName`; returns `false` otherwise.

hasNodeInTree(op, attributeName)

This function searches the process routing or operation sequence in which `op` occurs. It returns `true` if any node in the routing or sequence has the attribute with the specified name; it returns `false` otherwise.

Note that the search includes optional nodes, regardless of whether they are active in the process routing or operation sequence. To determine if an optional node is active, check the value of its `inclusionStatus` property (see `InclusionStatus` and Examples).

hasNodeInTreeWithTrueValue(op, attributeName)

This function searches the process routing or operation sequence in which `op` occurs. It returns `true` if any node in the routing or sequence has the attribute with the specified name and that attribute evaluates to `true`; it returns `false` otherwise.

Note that the search includes optional nodes, regardless of whether they are active in the process routing or operation sequence. To determine if an optional node is active, check the value of its `inclusionStatus` property (see `InclusionStatus` and Examples).

Examples

Following is an example of checking a simple hole to see if its operation sequence includes a hole finishing operation. The check is performed by feasibility modules of primary holemaking operations. If a finishing operation is included in the sequence, feasibility doesn't require that the primary holemaking operation be capable of achieving, by itself, the GCD's desired tolerance. (The library function `IsProcessAllGtolCapable2()` tests whether the primary operation alone can achieve the required tolerance.)

Note that optional nodes always return true when you use `hasNodeInTree()`, regardless of whether they are ultimately evaluated. To determine if an optional node is active, use `getNodeInTree()` to retrieve the node in question, and then check the value of its `inclusionStatus` property, as in the example below.

```
IsProcessAllGtolCapable(gcd, process) = {
  true if not(gcd.isAnyToleranceSpecified)
  true if hasAttribute(op, 'primaryHolemakingOp') and _
  (safeEval(getNodeInTree( op, 'SimpleHoleFinishing'
  ).inclusionStatus, _
  null) == InclusionStatus.AUTO_INCLUDE or _
```

```

safeEval(getNodeInTree( op, 'SimpleHoleFinishing'
).inclusionStatus, _
    null) == InclusionStatus.USER_INCLUDE)
true if hasAttribute(op, 'primaryGtolOp') and _
(safeEval(getNodeInTree( op, 'FinishGrinding' ).inclusionStatus,
_
    null) == InclusionStatus.AUTO_INCLUDE or _
safeEval(getNodeInTree( op, 'FinishGrinding' ).inclusionStatus, _
    null) == InclusionStatus.USER_INCLUDE)
IsProcessAllGtolCapable2(gcd, process) otherwise }

```

The following example illustrates working with parent/child GCD relationships. It considers a child Planar Face of a Ring, and uses `getNodeInTree()` to check whether heat treatment is included in the component's routing (the parent's parent's process routing). This approach also works with basic `getAttributeValue()` functions.

```

(safeEval( _
    getNodeInTree( _
        op.parentArtifactResult.parentArtifactResult, _
        'heatTreatProcess' ).inclusionStatus, _
    null) == InclusionStatus.AUTO_INCLUDE or _
safeEval( _
    getNodeInTree( _
        op.parentArtifactResult.parentArtifactResult, _
        'heatTreatProcess').inclusionStatus, _
    null) == InclusionStatus.USER_INCLUDE)

```

The following example is used in Casting, and is related to yields. aPriori starting point models assume that any parts scrapped within the Casting process group (that is, within a casting foundry) can be remelted. So the starting point calculates two different versions of final yield. One is based on the number of parts scrapped both within Casting and by any secondary processes (this is used to calculate overhead costs for Casting). The other is based on the number of parts scrapped outside of the foundry, that is, by secondary processes only (this is used to calculate material cost in Casting).

The code below checks to see if the node in question has the `inFoundryProcess` attribute, which indicates that the process occurs within the Casting foundry (where scrapped parts are remelted).

```

set {
    global.numScrapPartsDownStream = {
        numScrapParts if safeEval(global.numScrapPartsDownStream, null)
        == null
    }
}

```

```

numScrapParts + global.numScrapPartsDownStream otherwise }

global.numScrapPartsOutsideFoundry = {
  _LEAVE_UNCHANGED_ if hasAttribute( op, 'inFoundryProcess' )
numScrapParts if safeEval(global.numScrapPartsOutsideFoundry,
null) == null
numScrapParts + global.numScrapPartsOutsideFoundry otherwise }
}

```

The following example illustrates how to populate Machining custom outputs that support the display of total times for holemaking, roughing, and finishing. Each machining operation is tagged with the node attribute `operationCategory` whose values is Roughing, Finishing, or Holemaking. The library `libCustomProcessOutputs.csl` runs through all the operation cycle times and categorizes them based on the operation category.

```

ca_totalSurfaceFinishingOperations3 = _
  select sum(op.formulaResults.cycleTime) from allSetupOps op where
  -
    op.formulaResults.cycleTime != null and _
      getAttributeValue( op, 'operationCategory' ) ==
        'Finishing'

```

Following is an example that uses a node attribute to tag an operation so that other operations can determine whether the tagged operation is present in the current operation sequence. This example checks whether a Simple Hole has been previously threaded, where other processes in the routing that perform threading are tagged with the node attribute `PreviouslyThreaded`. You'll find this attribute on operations such as Simple Hole Tapping in Progressive Die.

```

Rule IsThisThreaded: gcd.threaded == true and _
  not(hasNodeInTree(op, 'PreviouslyThreaded'))

Message IsThisThreaded: _
  'This hole is unthreaded or threaded by a prior operation'

```

The following example uses `getNodeInTree()` together with `getAttributeValue()` to check a machining operation sequence for the presence of a finishing operation:

```

foo = getNodeInTree(op, 'operationCategory')

Rule foo: getAttributeValue(foo, 'operationCategory') == 'Finishing'

```

Note that if there are multiple nodes with the specified attribute, `getNodeInTree()` returns only the *first* one.

Routing Navigation Functions

isNodePrecededBy(node, precedingNodeName)

Returns `true` if `node` is preceded by `precedingNodeName` in the current process routing or operation sequence; returns `false` otherwise.

example:

```
bool = isPrecededBy(op, 'Drilling')
```

Error Handling Functions

assert(val, ruleName)

Tests the rule with the specified name. If it evaluates to `true`, `val` is returned. If the rule evaluates to `false`, a costing exception is thrown (halting costing at that level). If a message is associated with the rule that failed, that message is used in the costing exception.

assert(val, ruleName, message)

Tests the rule with the specified name. If it evaluates to `true`, `val` is returned. If the rule evaluates to `false`, a costing exception using `message` is thrown (halting costing at that level).

fail(message)

Throws a costing exception using `message` (halting costing at that level).

msg(x, y, ...)

Takes one or more arguments and returns a string that is the concatenation of the string form of each argument. Decimal precision is limited to four digits after the decimal. This function can be nested inside `fail()` or `assert()` calls to improve messaging.

Examples:

```
message = fail('Literal')
message2 = fail(indirect)

indirect = 'Indirect'

//bar throws an exception with the message 'Zero val indirect'
bar = {
  fail(zeroval) if (1 > 0)
  3 otherwise
}

//message3 causes a failure whose message is "The diameter -1 is bogus."
```

```

message3 = fail(msg('The diameter: ', diameter, ' is bogus.'))
diameter = -1

//assert tests
atruel = assert(foo, myrule) //Returns 7
atruemsg = assert(foo, myrule, 'Should pass') //Returns 7
atruemsgindirect = assert(foo, myrule, shouldpass) //Returns 7
foo = 7
shouldpass = 'Should pass indirect'
Rule myrule: 1 > 0

afalse = assert(foo, myfailrule) //Throws an exception with a default
message

//Throws an exception with the message 'Zero val'
afalsemsg = assert(foo, myfailrule, 'Zero val')

//Throws an exception with the message 'Zero val indirect'
afalsemsgindirect = assert(foo, myfailrule, zeroval)

zeroval = 'Zero val indirect'

Rule myfailrule: 1 < 0

inline = 3 + assert(foo, myrule, shouldpass) //Returns 10

//Throws an exception with the message 'Zero val indirect'
inlinefail = 3 + assert(foo, myfailrule, zeroval)

// Compute Cycle Time for Each Bend

//operation cycletime in secs
cycleTime = machine.bendCycleTime + manipulationTimePerBend

cycleTime = fail('Test fail.')

finishMass = part.volume * material.density * 1e-9

//use finish part mass as we can assume that bending occurs after
//all material removed by other ops
partMass = finishMass

// time (secs) for load/unload etc within the process cycle (secs)
manipulationTimePerBend = _
  select first(entry.manipulationAllowance) _
    from smBendBrakeHandling entry _
  where entry.weight >=partMass _
  order by entry.manipulationAllowance

```

Interpolation Function

interpolate(x, x0, x1, y0, y1)

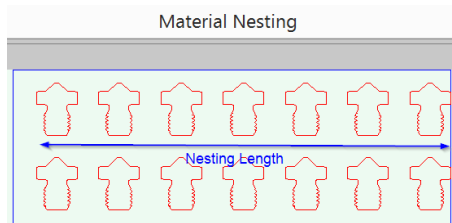
Returns the y-coordinate of the point on the specified line whose x-coordinate is *x*. The specified line is the line containing both the points (*x0*, *y0*) and (*x1*, *y1*). If *x0* equals *x1*, the function returns *y0*.

Miscellaneous Functions

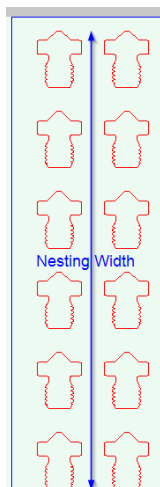
calculateNesting(op)

Given an operation on an Additive Manufacturing part, returns an object that represents the arrangement of parts nested on the build platform. This object has the following fields:

- **numberOfPartsPerLayer:** number of parts on the build platform, or (for DMLS), the number of parts in one layer of parts in the build chamber.
- **actualNestingLength:** length of the portion of the build platform that is occupied by the nested parts.



- **actualNestingWidth:** width of the portion of the build platform that is occupied by the nested parts.



This function returns a value for operations in both the Additive Manufacturing and Machining process groups.

dtcMessage(field-name-1, field-value-1, ... field-name-n, field-value-n)

This function should be used in conjunction with a CSL Advice construct, as described in [Advice](#).

A call to the function creates a **dtcMessage** object. When the cost engine evaluates the Advice construct that includes the call, it adds the created **dtcMessage** object to the list of **dtcMessage** objects associated with the current process routing. For a given costing, this list of **dtcMessage** objects for the selected routing determines the contents of many of the Design-to-Cost displays for Sheet Metal, Casting, and Plastic Molding.

The arguments specify the values of some of the fields of the created **dtcMessage** object. The arguments consist of alternating field names and field values. A field value

argument specifies the value for the field named by the argument that precedes it. Here is a sample call to the function in the context of an Advice Rule and Advice construct:

```
AdviceRule holeToEdgeMessage: badHoleEdgeRelations <> null
Advice holeToEdgeMessage: dtcMessage(
  'attributeld', 'dtcHoleAdvice.holeToEdgeAttribute',
  'messageld', 'dtcHoleAdvice.holeToEdgeMessage',
  'secondaryGcd', blankGcdKeyName,
  'current', worstEdgeRelation.distance,
  'suggestedMin', holeToBlankRecommendedProcess,
  'unitType', DTC_LENGTH,
  'custom.edgeList', listAsString(holeToEdgeNameListDistinct),
  'category', DTC_PROXIMITY)
```

A field is either predefined or user-defined. The predefined fields are listed below. A user-defined field must have a name that starts with “**custom.**”, for example, “**custom.msgToIType**”.

The value of a predefined field is either user-specified or automatically-supplied. User-specified values are specified by arguments to the **dtcMessage** function. Values that are supplied automatically are based on the runtime context or on user-specified values for other fields.

Following are the predefined fields that are user-specified:

- **attributeld**: string specifying the attribute or aspect of the current GCD that played a key role in triggering the associated advice rule or in causing the failure of the associated routing rule. This string is specified in one of two ways:
 - Directly, as in '**Diameter**' (for a rule that failed, for example, because the current GCD's diameter is too small)
 - Indirectly as a lookup key, such as '**dtcHoleAdvice.holeMinAttribute**'. The key, prefixed with '**DtcMessages**' (as in '**DtcMessages.dtcHoleAdvice.holeMinAttribute**'), is used to look up a message in the properties file **cmvMessages.properties**.

The string specified by this field is used to populate the following elements of Design-to-Cost displays:

- **Tolerance** column on the Finishing Operations tab of the Tolerances and Special Finishing Operations dialog.
 - **Issue** dropdown menu for the Casting Issues dialog (which appears when you click **Review** in the Casting section of the Design-to-Cost panel).
 - **Issue** dropdown menu for the Plastic Issues dialog (which appears when you click **Review** in the Plastic Issues section of the Design-to-Cost panel).
 - **Sub-Type** column for the Fabrication Issues dialog (which appears when you click **Review** in the Fabrication Issues section of the Sheet Metal Design-to-Cost panel).
- **messageld**: string specifying the circumstances that triggered the associated advice rule or that caused the failure of the associated routing rule. This string is specified in one of two ways:

- Directly, as in **'Hole is too small'** (for a rule that failed, for example, because the current GCD's diameter is too small)
- Indirectly as a lookup key, such as **'dtcHoleAdvice.holeMinMessage'**. The key, prefixed with **'DtcMessages'** (as in **'DtcMessages.dtcHoleAdvice.holeMinMessage'**), is used to look up a message in the properties file **cmvMessages.properties**.

The specified string can include placeholders that reference other fields, including custom fields and read-only fields. A placeholder for a given field is the name of the field enclosed in braces and prefixed by \$. Here are some examples:

'Hole is too small to be cut with \${process}'

'Hole is too close to the following hole(s): \${custom.holesList}'

The read-only field **messageText** contains the result of substituting the appropriate values for the placeholders.

The string specified by this field provides the text for the message that appears below the table in the Casting Issues, Plastic Issues, or Fabrication Issues dialog when a table row is selected.

- **secondaryGcd**: GCD, other than the current GCD, that played a key role in triggering the associated advice rule or in causing the failure of the associated routing rule. For example, if an advice rule is triggered because the current hole is too close to a second hole, **secondaryGcd** might be set to the second hole.

The value of this field is used to populate the **GCD #2** column of the Proximity Issues table in the Fabrication Issues dialog (for the Sheet Metal process group).

- **current**: this is the value of an attribute of the current GCD that played a key role in triggering the associated advice rule or in causing the failure of the associated routing rule. For example, if an advice rule is triggered because the current hole is too close to a second hole, **current** might be set to the distance between the holes.

The read-only field **currentText** provides a display-friendly version of the value of **current**, for example, **"10 mm"** or **"15 °"**. The text includes the default units associated with the unit type specified by the **unitType** field. See also **decimalPlaces**, below.

The value of the **currentText** field is used to populate the **Current** column of the table in the Casting Issues, Plastic Issues, or Fabrication Issues dialog. It is also used to populate the **Current value** column of the table in the Finishing Operations tab of the Tolerances and Machining dialog.

- **suggested**: this is an advisable value for the attribute whose actual value is given by **current**. Set this field if advice is triggered because **current** differs from the advisable value. See also **suggestedMin** and **suggestedMax**, below.
- **suggestedMin**: this is the minimum advisable value for the attribute whose actual value is given by **current**. Set this field if creation of advice is triggered because **current** is less than the advisable minimum (or, more generally, because **current** is not between the advisable minimum and advisable maximum—see **suggestedMax**, below). For example, if an advice rule is triggered because the current hole is too close to a second hole, **suggestedMin** might be set to the minimum advisable distance between holes.

- **suggestedMax**: this is the maximum advisable value for the attribute whose actual value is given by **current**. Set this field if advice is triggered because **current** is greater than the advisable maximum (or, more generally, because **current** is not between the advisable minimum and advisable maximum—see **suggestedMin**, above). For example, if an advice rule is triggered because the current hole is too deep, **suggestedMax** might be set to the maximum advisable hole depth.
- **unitType**: unit type for the values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**. This field is used to help provide display-friendly versions of these values via the read-only fields **currentText** (see **current**, above) and **suggestedText** (see below).
- **decimalPlaces**: number of decimal places to use for the display of values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**. This field is used to help provide display-friendly versions of these values via the read-only fields **currentText** (see **current**, above) and **suggestedText** (see below). If the field is not set or is set to -1, a maximum of 3 decimal places are displayed. The table below shows several examples of **currentText** values for various combinations of **decimalPlaces** and **current**:

decimalPlaces	current	currentText
-1 (default)	5	5
-1 (default)	5.3	5.3
-1 (default)	5.78264	5.783
2	5	5.00
2	5.78264	5.78

- **category**: free-form string that is used to categorize advice messages. For example, the value of **category** is **DTC_PROXIMITY** (defined as the string 'proximityWarning') for any **dtcMessage** object that is used to report a proximity issue in the Fabrication Issues dialog for the Sheet Metal process group.
- **priority**: relative importance of this advice message. In a future release, Design to Cost may support filtering of messages by importance.

An invocation of the **dtcMessage** function can contain 0 or more field-name/field-value pairs. But note that a **dtcMessage** object is used to determine the content of Design-to-Cost panels only if the appropriate fields are set. See further below.

Following are the read-only fields of **dtcMessage** objects. Their values are set based on field values specified by function arguments (see for example the field **current**, above) or based on aspects of the context in which the function is invoked, such as the current operation or GCD.

- **gcd**: current GCD.
- **process**: current process or operation.
- **vpe**: current VPE.
- **attributeText**: this is the string specified by the field **attributeId** (see above). When the **dtcMessage** function is executed, the cost engine attempts to use the value of **attributeId** (prefixed with 'DtcMessages') as a lookup key. If a message with that key is found in **cmvMessages.properties**, the field **attributeText** is set to that message. Otherwise, **attributeText** is set to the value of **attributeId**.

- **messageText**: this is the string specified by the field **messageld** (see above). When the **dtcMessage** function is executed, the cost engine attempts to use the value of **messageld** (prefixed with 'DtcMessages') as a lookup key. If a message with that key is found in **cmvMessages.properties**, the field **messageText** is set to that message. Otherwise, **messageText** is set to the value of **messageld**.
- **currentText**: display-friendly version of the field **current** (see above), for example, "10 mm" or "15 °". The text includes the default units associated with the unit type specified by the **unitType** field (see above). See also **decimalPlaces**, above.

The value of the **currentText** field is used to populate the **Current** column of the table in the Casting Issues, Plastic Issues, or Fabrication Issues dialog. It is also used to populate the **Current value** column of the table in the Finishing Operations tab of the Tolerances and Machining dialog.

- **suggestedText**: display-friendly version of **suggested**, if **suggested** is set. Otherwise, **suggestedText** is a display-friendly version of **suggestedMin** and/or **suggestedMax**, if one or both is set. The text assumes the default units of the type specified by the field **unitType**, and limits the number of decimal places displayed to the value specified by the field **decimalPlaces** (see above).

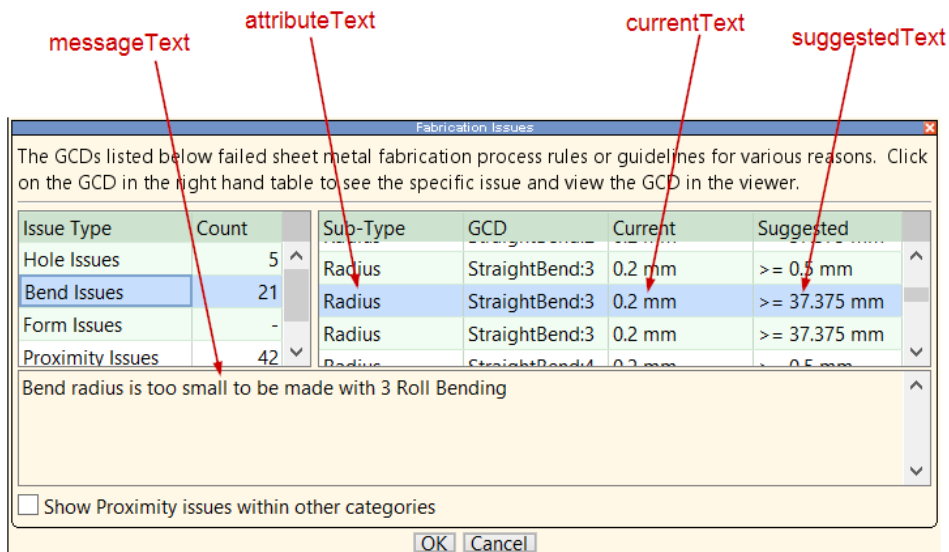
The table below shows several examples of **suggestedText** values for various combinations of **suggested**, **suggestedMin**, and **suggestedMax**:

suggested	suggestedMin	suggestedMax	unitType	suggestedText
5			Angle	5 °
	3		Length	>= 3 mm
		8	Length	<= 8 mm
	2	18	Length	2 – 18 mm
10	15	30	Angle	10°
12				12
	2	5		2 – 5

If you want to display advice in the **Fabrication Issues** dialog for the Sheet Metal process group, set some or all of the following fields in your call to the **dtcMessage** function (all fields are optional except **category**):

- **attributeld**: specify the string that you want to appear in the **Sub-Type** column. Not used for Proximity Issues.
- **messageld**: specify the string that you want to appear below the table when the corresponding table row is selected. The string that appears is the result of substituting the appropriate values for placeholders in the string specified by **messageld**.
- **secondaryGcd**: specify the GCD whose name you want to appear in the column **GCD #2**. This is generally the GCD to which the current GCD is too close, or to which the current GCD bears some problematic proximity relation. Only used for Proximity Issues.
- **current**: specify the value that you want to appear in the **Current** column. The string that appears includes the default units for the type specified by **unitType**, and uses, at most, the number of decimal places specified by **decimalPlaces**.

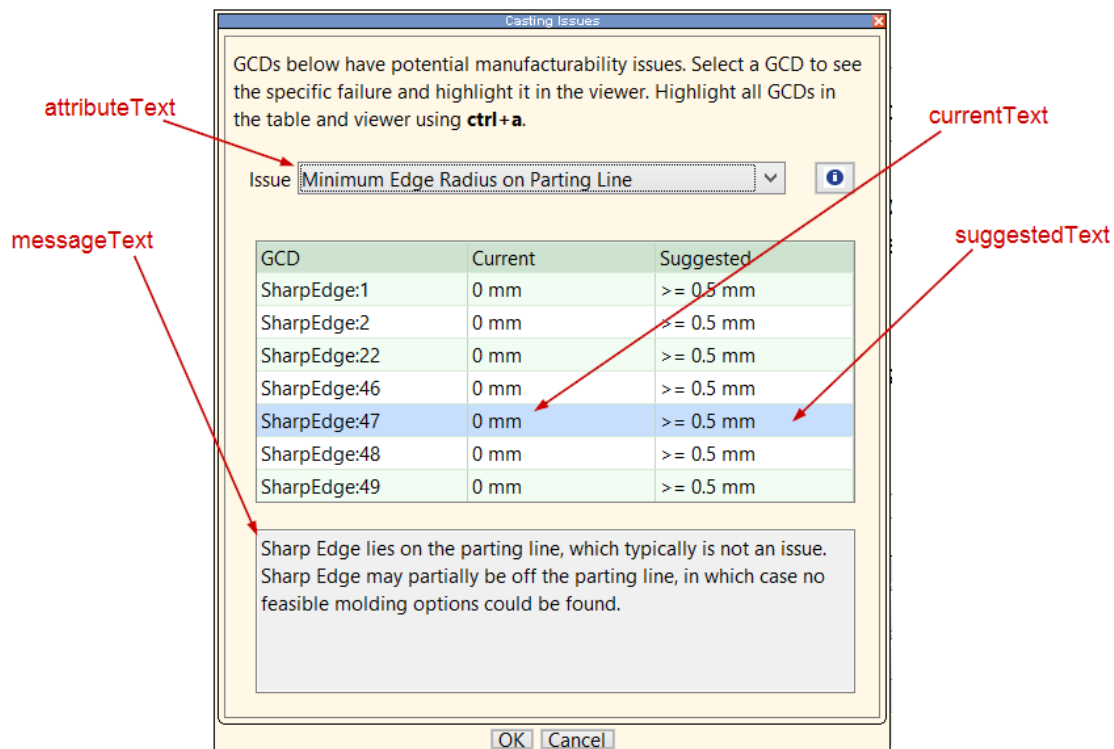
- **suggested**, **suggestedMin**, **suggestedMax**: specify the value or range that you want to appear in the **Suggested** column. The string that appears is the value of **suggestedText**—see above.
- **unitType**: set this to the unit type that is associated with the values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.
- **decimalPlaces**: set this to the maximum number of decimal places to use for the display of values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.
- **category**: set this to one of the following, depending on what **Issue Type** you want your advice to fall under:
 - 'proximityWarning'
 - 'bendIssue'
 - 'formIssue'
 - 'materialIssue'
 - 'holeIssue'



If you want to display advice in the **Plastic Issues** or **Casting Issues** dialog, set some or all of the following fields in your call to the **dtcMessage** function (the fields **attributeld** and **custom.panelOutput** are required; other fields are optional):

- **attributeld**: specify the text for the menu-item that you want the user to select in order to display the advice.
- **messageld**: specify the string that you want to appear below the table when the corresponding table row is selected. The string that appears is the result of substituting the appropriate values for placeholders in the string specified by **messageld**.
- **current**: specify the value that you want to appear in the **Current** column. The string that appears includes the default units for the type specified by **unitType**, and uses, at most, the number of decimal places specified by **decimalPlaces** (see **currentText**, above).

- **suggested**, **suggestedMin**, **suggestedMax**: specify the value or range that you want to appear in the **Suggested** column. The string that appears is the value of **suggestedText**—see above.
- **unitType**: set this to the unit type that is associated with the values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.
- **decimalPlaces**: set this to the maximum number of decimal places to use for the display of values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.
- **custom.panelOutput**: set this to one of the following:
 - 'castingIssue'
 - 'plasticIssue'



If you want to display advice in the **Tolerances and Machining** dialog, set some or all of the following fields in your call to the **dtcMessage** function (the fields **attributeId** and **category** are required; other fields are optional):

- **current**: specify the value that you want to appear in the **Current Value** column. The string that appears includes the default units for the type specified by **unitType**, and uses, at most, the number of decimal places specified by **decimalPlaces**.
- **suggested**, **suggestedMin**, **suggestedMax**: specify the value or range that you want to appear in the **Primary Process** column. The string that appears is the value of **suggestedText**—see above.
- **unitType**: set this to the unit type that is associated with the values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.
- **decimalPlaces**: set this to the maximum number of decimal places to use for the display of values of **current**, **suggested**, **suggestedMin**, and/or **suggestedMax**.

- **category:** set this to one of the following:
 - 'toleranceWarning'
 - 'roughnessWarning'
 - 'roughnessRzWarning'
 - 'diamToleranceWarning'
 - 'positionToleranceWarning'
 - 'circularityWarning'
 - 'concentricityWarning'
 - 'cylindricityWarning'
 - 'parallelismWarning'
 - 'perpendicularityWarning'
 - 'runoutWarning'
 - 'totalRunoutWarning'
 - 'straightnessWarning'
 - 'symmetryWarning'
 - 'profileOfSurfaceWarning'
 - 'flatnessWarning'

GCD	Operation	Cycle Time, s	Tolerance	Current value	Basic Machining Limit	Primary Process Limit	Process
SimpleHole:1	Reaming	2.24	Diam Tolerance	0.05000 mm	0.05730 mm	0.05730 mm	Perforating
			Diam Tolerance	0.05000 mm	0.05730 mm	0.05730 mm	Punching

gcdNeedsCam(artifact, coDirectionalEpsilon)

Returns true if none of the `isAccessible` setup axes for `artifact` are within `coDirectionalEpsilon` degrees of the `MainFace`'s normal direction; returns false otherwise.

getAngleBetweenPartStripFlowAndSetupAxis(holeSetupAxis, normalVector, eastDirection)

Calculates the angle between the part strip flow direction and the setup axis passed in, which is typically derived from a hole on a pierce cam. The function has the following arguments:

- `holeSetupAxis`: setup axis whose angle is to be calculated
- `normalVector`: main surface's normal vector
- `eastDirection`: main surface's east direction

Note: If the setup axis' direction is within .01 degrees of the normal vector, `null` is returned (because the calculation becomes unstable as the hole's setup axis approaches the same direction as the normal).

getDistanceToPartFrontEnd(turningAxis, Gcd)

Returns the distance from `Gcd`'s centroid to the `turningAxis`' front face. The `turningAxis` front face is the face that faces in the same direction the `turningAxis` is pointing. If `Gcd` does *not* have a centroid (for example, if it is an edge), `null` is returned.

getFlattenedOps(op)

Returns a flattened collection of all of `op`'s costed child operations.

Example:

```
numberOfFacing = _
  select count(op) from flattenedOps op where _
  op.simpleName=='Facing'

flattenedOps = getFlattenedOps(op)
```

**getCamAssignments(
 holes,
 codirectionalEpsilon,
 minimumDistanceBetweenHoles,
 rows
)**

Returns a collection of `CamAssignment` objects that assigns each hole in `holes` to one cam. The argument `rows` is a collection of rows from a lookup table (such as **camUnitSizes** from the Sheet Metal process group) that specifies size information about the various types of cams to be assigned.

Holes are placed on the same cam (`CamAssignment` object) if all the following hold:

- All the holes' setup axes are codirectional, that is, the angle between their setup axes varies by at most by `codirectionalEpsilon`.
- All holes meet the distance apart requirement, that is, the outer edges of holes are at least `minimumDistanceBetweenHoles` apart.
- The rectangular size covered by all the holes is not greater than the size of the cam mounting surface (the product of **Cam Face Width** and **Cam Face Height** for the row that represents the cam). For example, two holes that are 1000mm apart are not both assigned to a cam that has a mounting surface that measures 500 X 500mm.

The function always chooses a cam with the smallest possible cam mounting surface possible for a group of holes. It also groups holes together until such a time as there is no cam mounting surface large enough for them, at which point subsequent holes are put on a new cam.

isTrue(string)

Returns `true` if the string passed in is non-null and equals (case insensitive) `'true'`, `'yes'` or `'1'`.

isNodePrecededBy(node, precedingNodeName)

Returns `true` if `node` is preceded by `precedingNodeName` in the current process routing or operation sequence; returns `false` otherwise.

example:

```
bool = isPrecededBy(op, 'Drilling')
```

safeEval(expression, defaultValue)

Returns the result of evaluating `expression`., if no error occurs during the evaluation. If any error occurs during that evaluation, `defaultValue` is returned.

For example, `safeEval(12*3, 2)` returns 36, while `safeEval(12 + bogus, 2)` returns 2, if `bogus` is not a defined expression or variable. See also Examples.

.

safeGet(x, altVal)

Returns `x` if `x` is non-null; returns `altVal` otherwise.

getSlidesAndLiftersForGCDs(gcDs, maxGCDPitch, maxActionLength, minimumLifterClearance, drawDirectionOrthogonalEpsilon)

Returns a collection with two elements (each of which is itself a collection):

- Collection of slides
- Collection of lifters

Each slide and lifter has the following fields:

- `start`: location relative to an arbitrary point. The length of the slide or lifter is the difference between `start` and `end`.
- `end`: location relative to an arbitrary point (the same one used by `start`). The length of the slide or lifter is the difference between `start` and `end`.
- `artifacts`: GCDs handled by this slide or lifter
- `setupAxis`: setup axis used by the GCDs handled by this slide or lifter

The function first determines what setup axis to use for each GCD, by doing the following:

- Sort GCDs based on size (volume), in descending order.
- For each GCD, iterate over each of its `isAccessible` relations.
- Check if any of the `isAccessible` relations has a distance to obstruction equal to -1, a setup axis that is within `drawDirectionOrthogonalEpsilon` degrees of being orthogonal to the draw direction, and a setup axis that has already been picked for a previous `gcd` for use in a Slide. If so, use that setup axis and assign this GCD to a slide. If more than one setup axis satisfies these criteria, use the one with the smallest length on the `isAccessible` relation. If more than one setup axis has the same smallest length, use the one with the smallest `distanceToSolidShadowBorder`. Note that the function considers the length to

- be the same if it is literally the same, or if the two setup axes for the two relations are within 5 degrees of each other or within 5 degrees of being exact opposite directions (180 degrees). This is because there is some latitude when length is populated and length for two setup axes pointing in the same direction or in opposite directions should essentially be the same.
- If no setup axis was chosen in the previous step, check if any `isAccessible` relations have a distance to obstruction greater than or equal to `(minimumLifterClearance + isAccessible.length)` or `distanceToObstruction=-1` with setup axis *not* within `drawDirectionOrthogonalEpsilon` degrees of being orthogonal to the draw direction and a setup axis that has already been picked for a previous GCD for use in a lifter. If so, use that setup axis and assign this GCD to a lifter. If more than one setup axis satisfies these criteria, use the one with the smallest length on the `isAccessible` relation.
 - If no setup axis was chosen in the previous steps and if there exists an `isAccessible` relation with distance to obstruction equal to `-1` and a setup axis that is within `drawDirectionOrthogonalEpsilon` degrees of being orthogonal to the draw direction, use the setup axis on that relation and assign this GCD to a slide. If more than one relation/setup axis pair satisfies these criteria, use the one with the smallest length on the `isAccessible` relation.
 - If no setup axis was chosen in the previous steps, check if any `isAccessible` relations have a distance to obstruction greater than or equal to `(minimumLifterClearance + isAccessible.length)` or `distanceToObstruction=-1` with setup axis *not* within `drawDirectionOrthogonalEpsilon` degrees of being orthogonal to the draw direction, use the setup axis on that relation and assign this GCD to a lifter. If more than one relation/setup axis pair satisfies these criteria, use the one with the smallest length on the `isAccessible` relation.
 - If no setup axis was chosen in the previous steps, put the GCD on its own lifter.

All the GCDs passed in are thus grouped together according to their setup axis as found in step 1 and whether they will be on a lifter or a slide.

For each group of GCDs with the same setup axis, a group of lifters or slides will be created. Lifters/slides are non-overlapping with a maximum length of `maxActionLength`. If the start of a current GCD falls within an existing lifter/slide's start + `maxActionLength`, then that GCD is added to that existing lifter. Otherwise, a new lifter/slide is created whose start is the GCD's start. If a GCD is assigned to a lifter/slide and the GCD's end runs past the lifter/slide's start + `maxActionLength`, then an additional lifter/slide is created and the GCD is assigned to that additional lifter/slide. In addition, if the end of one GCD minus the start of the next GCD is greater than `maxGCDPitch` then these two GCDs are placed on different lifters/slides. In addition, any GCD passed in that falls through to the last step above is placed on its own lifter. Note that this lifter will have no setup axis nor will it have a start/end.

getAssociatedSlides(gcd, slideList, forLengthWise)

Returns a pair of collections, each of which contains all of the slides that are allocated to either one side or the other side of the specified GCD. If `forLengthWise` is `true`, the function returns all slides that fall on either side of the lengthwise sides of the part. If `forLengthWise` is `false`, the function returns all slides that fall on either side of the

widthwise sides of the part. The four sides of the part are divided into equal 90 degree quadrants. Which quadrant (or which side) a slide gets assigned to depends on which quadrant contains its setup axis.

Example:

```
numSlidesLengthWiseSide2 = 
  select count(s) from slidesInLengthWiseDirection.second s

numSlidesWidthWiseSide1 = 
  select count(s) from slidesInWidthWiseDirection.first s

numSlidesWidthWiseSide2 = 
  select count(s) from slidesInWidthWiseDirection.second s

slidesInLengthWiseDirection = getAssociatedSlides(gcd, slides, true)

slidesInWidthWiseDirection = getAssociatedSlides(gcd, slides, false)
```

getAssociatedSlideBundles(gcd, slideBundleList, forLengthWise)

Returns a pair of collections, each of which contains all of the slide bundles that are allocated to either one side or the other side of the specified GCD. If `forLengthWise` is `true`, the function returns all slide bundles that fall on either side of the lengthwise sides of the part. If `forLengthWise` is `false`, the function returns all slide bundles that fall on either side of the widthwise sides of the part. The four sides of the part are divided into equal 90 degree quadrants. Which quadrant (or which side) a slide bundle gets assigned to depends on which quadrant contains the setup axis that the slide bundle is accessible from.

Example:

```
slideBundlesInLengthWiseSide1 = slideBundlesInLengthWiseDirection.first
slideBundlesInLengthWiseSide2 =
slideBundlesInLengthWiseDirection.second

slideBundlesInLengthWiseDirection = 
  getAssociatedSlideBundles(gcd, slideBundles, true)

slideBundles = select x from gcd.childArtifacts x where
isSlideBundle(x)
```

postMessageAndReturnValue(value, message)

Adds the string `message` to the message tree as an **Info** message (associated with a green dot). Returns `value`. To aid in internationalization, the actual parameter for `message` is typically a variable that is tied to a literal string within a centralized messages file.

Example:

```
postMessageAndReturnValue(2.0, successMsg) // in a file shared across
languages

successMsg = 'Calculation was successful' // in a language-specific
file
```

getClockAngle(normalDirection, noonDirection, clockDirection)

Returns the clock angle. Following is the calculation performed, in Java:

```
double cos = noonDirection.dot(clockDirection);
Vector3d cross = new Vector3d();
cross.cross(clockDirection, noonDirection);
double sin = normalDirection.dot(cross);
double angle = Math.atan2(sin, cos);
if (angle < 0) angle = 2 * Math.PI + angle;
return Math.toDegrees(angle);
```

hasField(obj, fieldName)

Returns `true` if the object has a field by the specified name; returns `false` otherwise.

getFinishedArea(plantRMillPercentile, gcd, op)

Returns the sum of all finished areas of all operations that lie on the specified GCD that have an `opDiameter` that is less than or equal to the *selected* `opDiameter`. The *selected* `opDiameter` is selected based on the `plantRMillPercentile` in conjunction with all the diameters of all operations that lie on the specified GCD.

Example:

```
GetAreaFinished3(millSize) = {
    getFinishedArea(plant.smallRMillPercentile, gcd, op) _
        if (millSize == SMALLEST_MILL)
    getFinishedArea(plant.largeRMillPercentile, gcd, op) _
        if (millSize == LARGEST_MILL)
    getFinishedArea(plant.middleRMillPercentile, gcd, op) _
        if (millSize == MIDDLE_MILL)
}
```

6 Common Task Examples

This chapter contains examples of several common CMWB tasks. All the examples center around a sample custom process, AbrasiveJet Cut in the Sheet Metal process group. AbrasiveJet Cut, as presented here, is almost identical to the starting point process Waterjet Cut. The examples in this chapter use AbrasiveJet Cut to illustrate how a process like Waterjet Cut might have been created, starting from a copy of a similar process, Laser Cut, in the same process group.

This chapter includes the following topics:


- Adding a New Process to a Process Group
 - Adding New Processes and Operations to Templates
 - Defining and Modifying Machine Types
 - Modifying Machine Selection
 - Adding Feasibility Rules
 - Adding New Plant Variables
 - Adding New Process Setup Options
 - Adding Lookup Tables
 - Modifying Taxonomy Modules
 - Adding and Modifying Library Modules
-

Adding a New Process to a Process Group

This section contains examples of adding a new process to a process group as a starting point for the full implementation of a custom process. The first procedure shows how to create an unpopulated process using the **File** menu. The second procedure shows how to copy an existing process to provide a more developed starting point. The other main steps involved in fully implementing a custom process are covered in subsequent sections of this chapter.

Creating a New Process from Scratch

Use this procedure if you want to create a completely new process with minimal starting data. This creates the costTaxonomy and machineSelectionRule CSL Modules, and populates them with the basic equations of a standard cost taxonomy, along with the necessary library file references to start configuring. Optionally you can also specify to create an empty machine table that has an identical format to one specified at creation time (no machines will be populated in the new table).

- 1 From the **File** menu, click **Create New Process...**
- 2 Follow the prompts in the resulting **Create New Process** window to provide:
 - VPE Name where the new process should be created.
 - Process Group where the new process should be created
 - The Name of the new process
 - (Optional) Whether or not to create an empty machine table with the same structure as the one specified
- 3 Click **OK** when done.
- 4 Continue to develop the new process using the later sections in this chapter.
- 5 When the process is complete, click **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Creating a New Process from an Existing Process

This example creates a custom process, AbrasiveJet Cut, which is copied from Laser Cut. AbrasiveJet will mimic the starting point process Waterjet Cut; here we copy from Laser Cut just as Waterjet Cut might have been created by starting from a copy of Laser Cut.

When you copy a process, you create copies of the original process's associated CSL modules, process setup options, data tables, and metadata tables. Optionally, you also create copies of the original process's associated operations (including their associated modules, setup options, data, and metadata).

The section includes the following subsections:

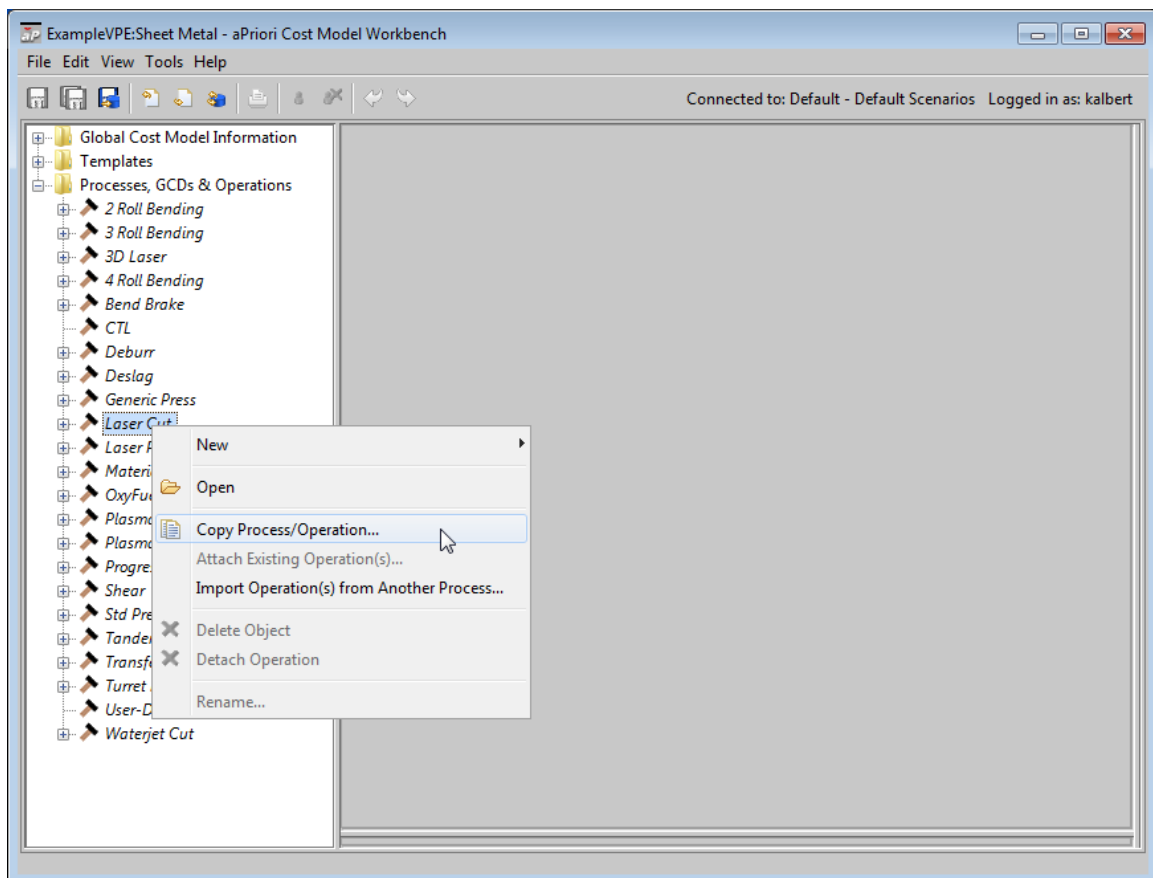
- Copying the Process
- Renaming the Operations
- Adding Operations

For more information on these tasks, see [Creating and Deleting Processes, Operations, and Branch Nodes](#) in the chapter [Working with Cost Model Logic](#).

Copying the Process

Follow these steps to make a copy of Laser Cut:

- 1 In the navigation pane, expand **Processes, GCDs & Operations**, right-click on **Laser Cut**, and select **Copy Process/Operation....**

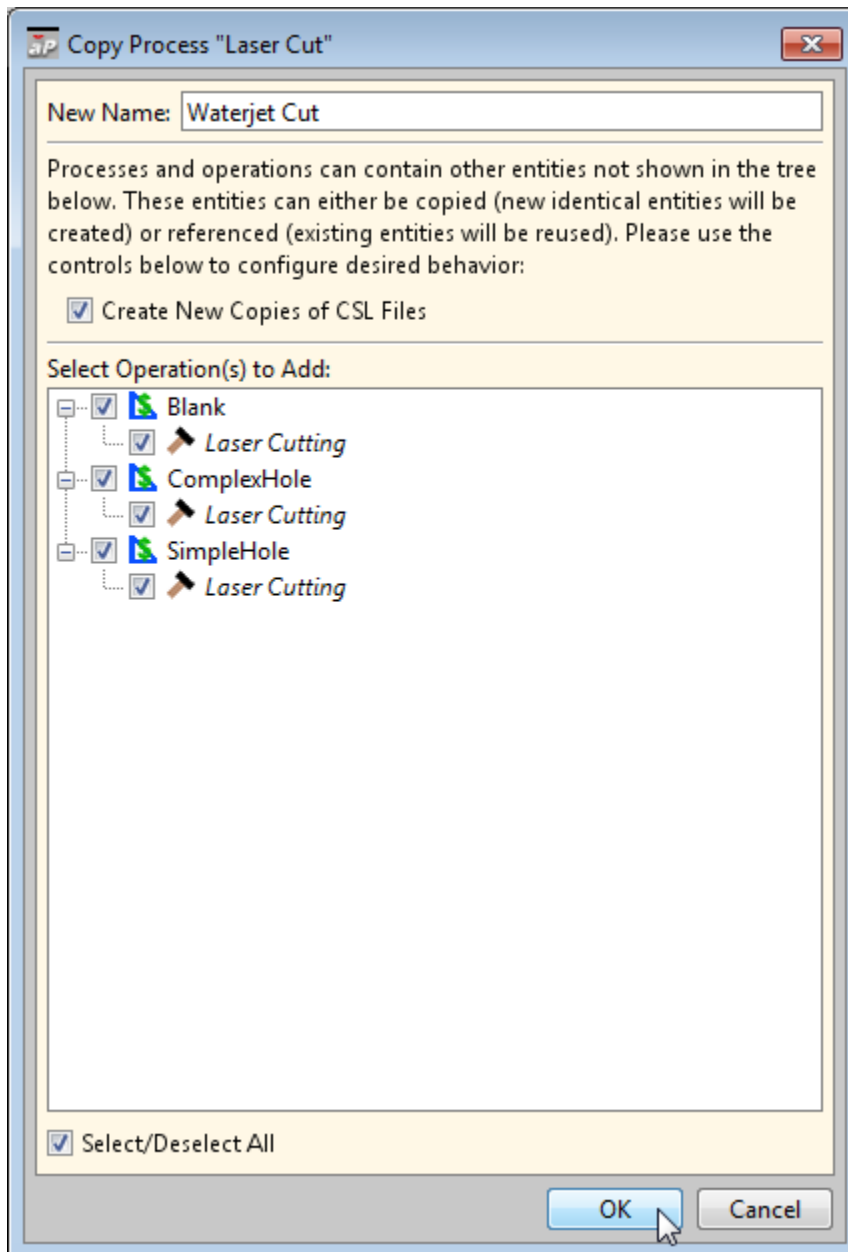



- 2 In the **Copy Process** dialog, ensure that all checkboxes are checked.

By checking **Create New Copies of CSL Files**, you ensure the creation of new copies of the CSL modules associated with the copied process. If you uncheck this checkbox, the copy will use all the same modules as the copied process.

By checking the other checkboxes, you ensure the creation of copies of the operations associated with the copied process. Following Waterjet Cut, the AbrasiveJet Cut process will have analogs of Laser Cutting operations for the GCDs Blank, ComplexHole, and SimpleHole. (It will also have operations for cutting Edge child GCDs of blanks and complex holes, but these are added in a separate step.)

- 3 In the **Copy Process** dialog, enter **AbrasiveJet Cut** in the **New Name** Field, and click **OK**.

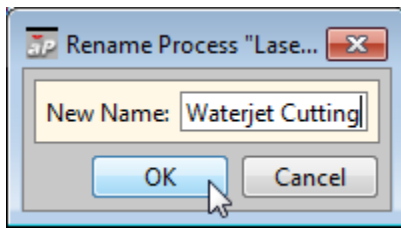



- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Renaming the Operations

For each operation copy, do the following:

- 1 In the navigation pane, right click on the operation, and select **Rename...**
- 2 In the Rename Process dialog, enter **Waterjet Cutting** in the **New Name** field.
- 3 Click **OK**.



- 4 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding Operations

To calculate cut times and pierce times for each GCD of the current part, Laser Cut uses operations on the blank, simple hole, and complex hole GCDs. As with Laser Cut, AbrasiveJet Cut uses an operation on the simple hole GCD to calculate pierce times and cut times for each simple hole. But unlike Laser Cut, AbrasiveJet Cut uses blank and complex hole operations to calculate pierce times only; AbrasiveJet Cut uses an operation on the edge GCD to calculate cut times for each edge of the blank and each edge of each complex hole. So we must add two new operations to AbrasiveJet Cut.

The following tables show which times are calculated by each processes and operation:

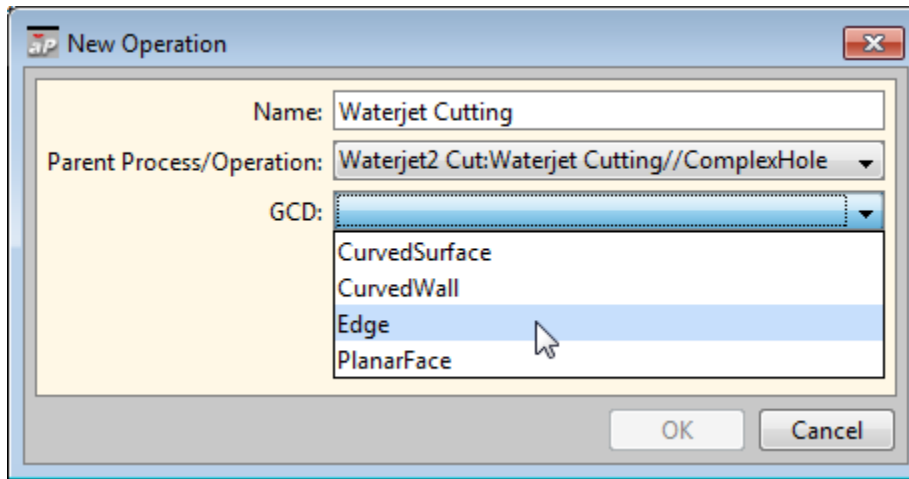
Laser Cut	Process	SimpleHole Operation	Blank Operation	ComplexHole Operation
Traverse Time	X			
Pierce Time		X	X	X
Cut Time		X	X	X


AbrasiveJet Cut	Process	SimpleHole Operation	Blank Operation	Blank Edge Operation	ComplexHole Operation	ComplexHole Edge Operation
Traverse Time	X					
Pierce Time		X	X		X	
Cut Time		X		X		X

For the AbrasiveJet Cutting operation under **Blank**, as well as for the AbrasiveJet Cutting operation under **ComplexHole**, do the following:

- 1 Right-click on the operation, and select **New > Operation...**

- 2 Resize the dialog box, if necessary, in order to make the operation pathnames entirely visible in the **Parent Process/Operation** field. Ensure that this field is set to the operation you right-clicked on.
- 3 In the **New Operation** dialog, click the **GCD** field and select **Edge**.



- 4 Click OK.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Adding New Processes and Operations to Templates

The example in this section shows how to modify the templates in the Sheet Metal process group in order to incorporate routings for the sample, custom process AbrasiveCut (see Adding a New Process to a Process Group).

When you add a new process to a cost model, you typically add associated operations as well. This section includes the following subsections:

- Navigating to and Modifying Templates
- Modifying the Process-level Routings
- Modifying the Operation-level Routings

For more information on templates, see [Working with Templates](#) in the chapter Working with Cost Model Logic.

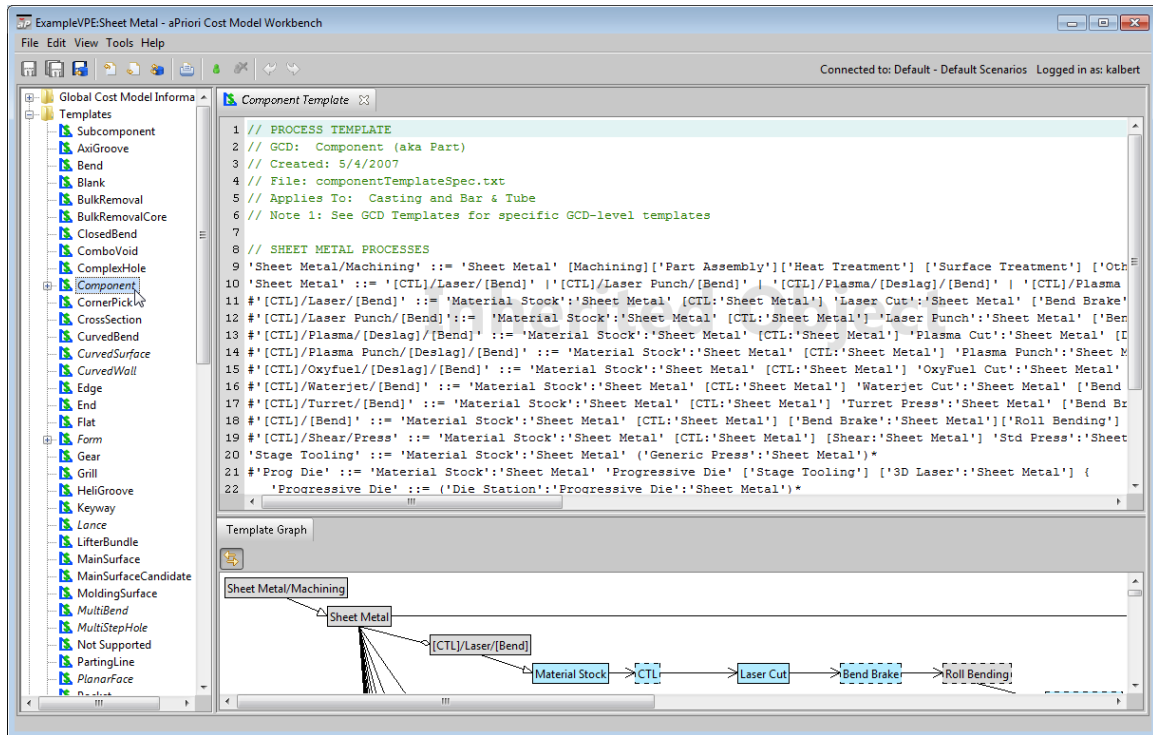
Navigating to and Modifying Templates




Follow these steps to navigate to and modify the template for a given GCD type:

- 1 In the navigation pane, expand Templates.

Double click the desired GCD type. The template text appears in the editing pane. (A graphical representation of the template may appear in the bottom part of the editing pane. **Show Info Panel** in the **View** menu controls its visibility. You may have to drag the

top border of the Info Panel up from near the bottom of the editing pane in order to make the Info Panel contents visible.)



- 2 Select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 3 Modify the text of the module in the editing pane.
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Modifying the Process-level Routings

This section shows you how to add process-level routings that are analogous to those of Laser Cut. Modify the AbrasiveJet Cut **Component** template as follows:

- 1 Find the line that starts with the following:
'Sheet Metal' ::=
- 2 In that line, select the following portion:
' [CTL]/Laser/[Bend] ' |
- 3 Type control-c to copy the selected text.
- 4 In that same line, click immediately before the following:
' [CTL]/Turret/[Bend] '

- 5 Type control-v to paste the copied text. This adds a new alternative routing to the top-level template node.
- 6 Replace `Laser` with `AbrasiveJet` in the newly pasted text portion.
- 7 Copy the line that starts with the following (copy the entire line):

```
# '[CTL]/Laser/[Bend]' ::=
```
- 8 Paste it as a new line. This line will define the alternative routing that you added in step 5.
- 9 In the new line, change `Laser` to `AbrasiveJet` throughout.
- 10 Find the line that starts with the following:

```
`Cutting' ::=
```
- 11 Click at the end of that line and enter a vertical bar, |, followed by the AbrasiveJet process node name:

```
| 'AbrasiveJet Cut': 'Sheet Metal'
```

You can copy the node name ('AbrasiveJet Cut': 'Sheet Metal') from the **Template Node Name** field of the **Node Attributes** tab for the AbrasiveJet Cut process.

Modifying the Operation-level Routings

This section shows you how to add operation-level routings that are analogous to those of Laser Cut, as well as a routing for the edge operation AbrasiveJet Cutting.

Modify the AbrasiveJet Cut **Blank**, **ComplexHole**, and **SimpleHole** templates as follows:

- 1 Find the line that starts with the following:

```
'Sheet Metal' ::=
```
- 2 In that line, click immediately before the following line:

```
'Turret Press': 'Turret Press': 'Sheet Metal'
```
- 3 Enter the operation node name followed by a vertical bar, |:

```
'AbrasiveJet Cutting': 'AbrasiveJet Cut': 'Sheet Metal' |
```

You can copy the node name from the **Template Node Name** field of the **Node Attributes** tab for the blank and complex hole AbrasiveJet Cutting operations.

This causes the blank, complex hole, or simple hole AbrasiveJet Cutting operation to be assigned to a part's blank, complex hole, or simple hole if the process AbrasiveJet Cut has been assigned to the part.

Modify the AbrasiveJet Cut **Edge** template as follows:

- 1 Add the following line:

```
'Sheet Metal' ::= 'AbrasiveJet Cut' [Machining] ['Part Assembly']
```
- 2 Below that, add the following, all on one line:

```
'AbrasiveJet Cut' ::=
  'AbrasiveJet Cut:AbrasiveJet Cutting//Blank:AbrasiveJet
Cutting//Edge' |
  'AbrasiveJet Cut:AbrasiveJet Cutting//
  ComplexHole:AbrasiveJet Cutting//Edge'
```

You can copy these node names from the **Fully-Qualified Name** field of the **Node Attributes** tab for the edge AbrasiveJet Cutting operations.

This causes the edge operation AbrasiveJet Cutting to be assigned to a part's edge if the blank, complex hole, or simple hole operation AbrasiveJet Cutting has been assigned to the edge's parent GCD (blank, complex hole, or simple hole). For more information on the edge operation, see [Adding Operations](#).

Defining and Modifying Machine Types

This section provides an example of defining a machine type for a new process. Once you've created or modified the machine type, you can use VPE Manager to create or modify the machine table that uses that type. Each line of the machine type table defines a machine attribute, and corresponds to a column of the machine table.

The example uses the sample custom process AbrasiveJet Cut (see [Adding a New Process to a Process Group](#)). The initial machine type definition mimics the starting point process Waterjet Cut in the Sheet Metal process group. This section also shows how to perform two modifications of this initial definition, in order to support further customization.



This section contains the following subsections:

- [Defining a Machine Type for a New Process](#)
- [Modifying a Machine Type—Padding Cycle Time](#)
- [Modifying a Machine Type—Preferring One Class of Machines Over Another](#)

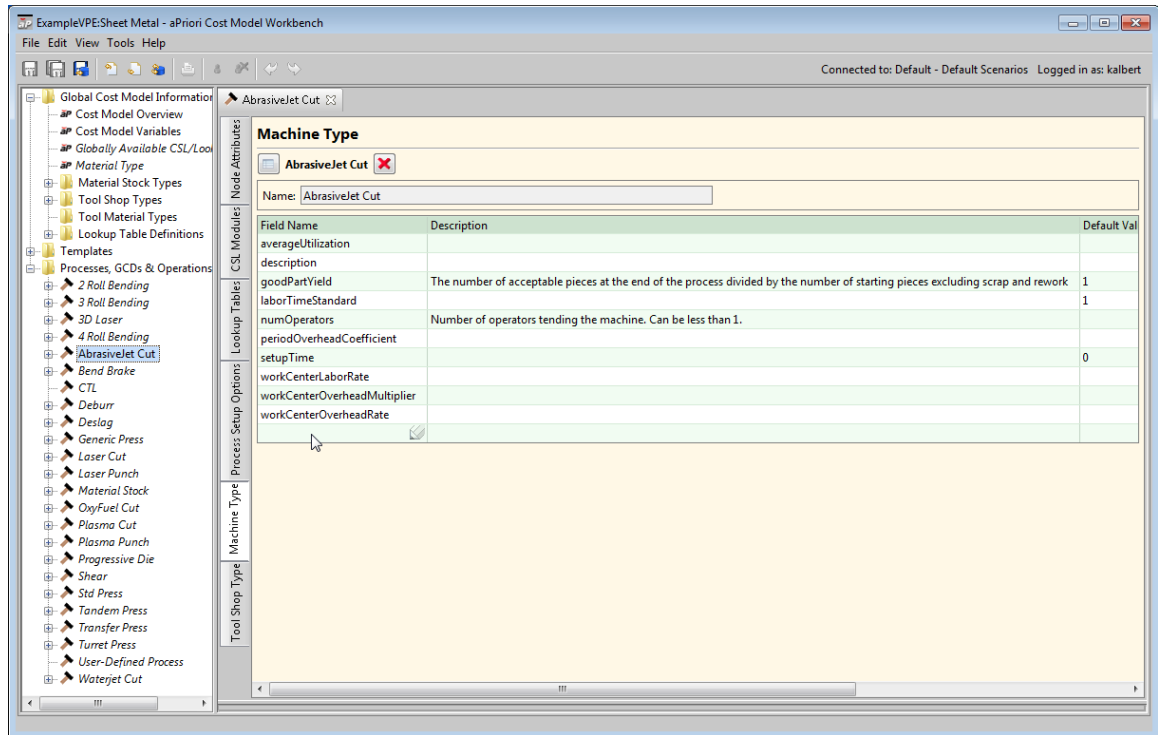
For more information on machine types, see [Working with Machine Metadata](#) in the chapter [Working with Cost Model Data and Metadata](#).

Defining a Machine Type for a New Process

This section shows how to create a machine type definition for AbrasiveJet Cut. Follow these steps to define a machine type for the new process:

- 1 Use the VPE Manager to delete all the machines associated with AbrasiveJet Cut. (When Laser Cut was copied, all its associated machines were copied.)
- 2 In the CMWB navigation pane, double click the AbrasiveJet Cut process. If the process is already open in the editing pane, close it and reopen it.
- 3 Click the **Machine Type** tab in the editing pane.
- 4 Click  to remove the table.
- 5 Click  to add a new, blank table.

- Type control-s to save the table and populate the new table with required fields (**averageUtilization**, **description**, **goodPartYield**, and so forth).



- Double click the Laser Cut process in the navigation pane, and click the **Machine Type** tab in the editing pane.
- Select all the rows of the Laser Cut machine type table, and type control C to copy the selected rows.
- Click the AbrasiveJet Cut tab at the top of the editing pane, click in the blank field at the bottom of the table, and type control-c to paste.
- Click the column heading **Field Name** to alphabetize.
- Remove duplicate rows. The required fields will each appear twice, once in editable form (marked with pencil icon) and once in non-editable form—the editable ones should be deleted.
- Add the rows in the table below. Double-click a cell to enter a value. In the **Unit Type Name** field, when you select an item from the dropdown list, a value for the **Unit** field appears automatically.

Field Name	Description	Default Value	Unit Type Name	Unit	Property Type name	Notes
abrasiveFlowRate	Flow rate of the abrasive into the mixing chamber.	.34	kg/min	kilogram / minute	double	

mixingTubeDiam	Diameter of the mixing tube.	.762	Length	millimeter	double	
nozzleDelay	Extra time required at pierce points to account for the nozzle turning on and off.	3.2	Time	second	double	
numNozzles	Number of nozzles on this machine available for cutting multiple parts at once.	1			double	
optimizationFactor	Factor used to distinguish between optimization software used by machine manufacturers.	1			double	
orificeDiam	Diameter of the jeweled orifice.	.356	Length	millimeter	double	
pressure		379.21	Pressure	newton / millimeter^2	double	


13 Remove the following rows:

- o numHeads
- o shuttleTime
- o toleranceFactor

14 In the **Default Value** column, add the following defaults:

- o numScrapsCutPerSheet: 2
- o sheetLengthTrimStrip: 5
- o sheetWidthTrimStrip: 5
- o smallFeatureFeedRadius: 12.7
- o smallFeatureThicknessRatio: 0



15 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.



16 Click the  icon to the left of the table name in the editing pane. The column properties and column groups page appears in a new tab in the editing pane.



The data table column names appear under **Available Fields** and **Displayed Fields**. These are the displayed names that correspond to the metadata table values of **Field Name**. Only those under **Displayed Fields** are displayed in VPE Manager and aPiori end user data tables.

To view and modify a column's properties, select the column name from **Available Fields** or **Displayed Fields**. The following fields appear under Column Properties

- **Name:** value of Field Name for this column.
- **Display Name:** displayed name of the column.
- **Formatter:** controls the format of displayed values.
- **Parent Group:** column group in which this column appears. In VPE Manager tables and aPriori end user tables, column groups can be expanded by clicking the plus sign.


To change which columns are displayed, select a column or group and use the left and right arrows,  and . Note that if you select a group and click the left arrow, all the columns in that group are moved out of **Display Fields**. Note also that whenever you move a column into **Display Fields**, you must re-specify its Parent Group.

To change the order in which columns appear in VPE Manager and aPriori end user tables, select a column or group, and use the up and down arrows,  and .

Create and delete column groups with  and .

Available Fields:	Displayed Fields:
Description Machine Type Machine Type Name Work Center Name Power	Primary ID Name Other ID Workcenter Optimization Factor Accounting Labor Rate Overhead Rate Overhead Multiplier Period Overhead Coefficient Number of Operators Labor Time Standard Time Setup Time Nozzle Delay Rates ▶ Rapid Traverse Rate ◀ Pressure Limits Bed Length Bed Width Small Feature Feed Radius Small Feature Thickness Ratio Number of Nozzles Other Orifice Diameter Mixing Tube Diameter Abrasive Flow Rate Scrap Cuts per Sheet Sheet Length Trim Strip Sheet Width Trim Strip Yields Avg Utilization Good Part Yield
Column Group Controls + x	
Column Properties Name: <input type="text"/> Display Name: <input type="text"/> Formatter: <input type="text" value="v"/> Parent Group: <input type="text" value="v"/>	

17 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.




18 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Modifying a Machine Type—Padding Cycle Time

If the use of some machines is associated with a known constant contribution to cycle time, you can define a machine attribute to support the specification of this constant. Taxonomy modules can then access the current machine in order to add the value to cycle time (see [Padding Cycle Time by Adding a Constant](#)). See also [Adding a Setup Option—Padding Cycle Time](#) and [Adding Plant Variables—Padding Cycle Time](#).

For the sample, custom process AbrasiveJet Cut, follow these steps to add such an attribute to the machine type:

- 1 Double click the AbrasiveJet Cut process in the navigation pane, and click the **Machine Type** tab in the editing pane.

In this example, the process is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 2 Click in the empty **Field Name** column of the last line of the table.
- 3 Enter the following information in the following columns:
 - **Field Name:** `cycleTimeAdditiveAdjustment`
 - **Default Value Text:** `0`
 - **Unit Type Name:** `Time`
 - **Property Type Name:** `double`
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.


See [Padding Cycle Time by Adding a Constant](#) for an example of using this attribute in a CSL taxonomy module.



Modifying a Machine Type—Preferring One Class of Machines Over Another

If you want machine selection for a particular process to prefer on-site machines to machines located at a remote facility, for example, you can define a string-valued machine attribute called, for example, `isOnSite`, to support such a selection requirement. The process's machine selection module can then access this attribute in order to distinguish between on-site and off-site machines (see [Preferring One Class of Machines Over Another](#)).

For the sample, custom process AbreasiveJet Cut, follow these steps to add such an attribute to the machine type:

- 1 Double click the AbrasiveJet Cut process in the navigation pane, and click the **Machine Type** tab in the editing pane.

In this example, the process is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 2 Click in the empty **Field Name** column of the last line of the table.
- 3 Enter the following information in the following columns:
 - **Field Name:** `isOnSite`
 - **Default Value Text:** `false`

- **Property Type Name:** `string`
- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
 - 5 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Preferring One Class of Machines Over Another](#) for an example of using this attribute in a CSL machine selection module.

Modifying Machine Selection

This section shows how to modify the machine selection module for AbrasiveJet Cut (which is copied from Laser Cut—see [Adding a New Process to a Process Group](#)), so that it mimics Waterjet Cut. The section also illustrates further customizing machine selection to prefer user-defined machines.

This section includes the following subsections:

- [Removing unwanted machine checks](#)
- [Preferring One Class of Machines Over Another](#)

For more information on machine selection, see [Machine Selection](#) in the chapter [Working with Cost Model Logic](#).

Removing unwanted machine checks

Suppose that we want any machine selected for AbrasiveJet Cut to pass the following two checks in order to ensure that the current part fits on the selected machine:

- Maximum part width check: blank width is less than or equal to the machine's maximum bed width.
- Maximum part length check: blank length is less than or equal to machine's maximum bed length.


In addition, let's say that we want machine selection to do the following:

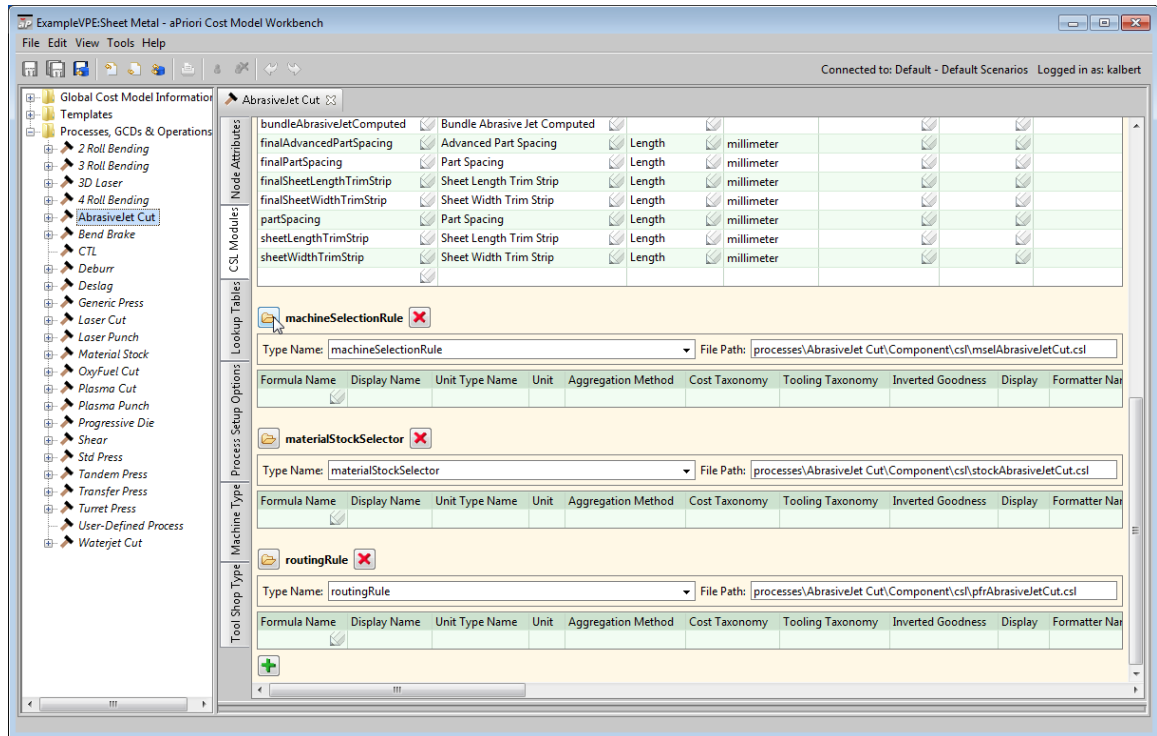
- Select the default machine if it passes these checks.
- Select the machine with the lowest overhead that passes the tests, if the default machine does not pass these checks.

Laser Cut machine selection is very similar. The difference is that Laser Cut machine selection includes two additional checks, a maximum thickness check and a minimum thickness check. AbrasiveJet Cut has related checks, but they are performed in the feasibility module, since they depend on criteria that are independent of any particular machine's characteristics (see [Adding Feasibility Rules](#)).


Follow these steps to modify machine selection for the AbrasiveJet Cut process in order to eliminate the unwanted checks:

- 1 In the navigation pane, expand **Processes, GCDs & Operations** and double-click the AbrasiveJet Cut process.

- 2 In the editing pane, select the CSL tab, and click on the folder icon, , next to **machineSelectionRule**.



The module text appears in the editing pane.

In this example, the module is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

- 3 Remove `and MaxThicknessCheck` and `MinThicknessCheck` from the query expression at the beginning of the module, resulting in the following formula:

```
machine = select first(m) from machines m _
  where MaxPartWidthCheck and MaxPartLengthCheck _
  order by isDefaultMachine(m) desc, overheadCost(m)
```

This query orders the feasible machines by the result of `isDefaultMachine` (a function defined below the query in this module). The result of `isDefaultMachine` is either `true` or `false`. In the ascending order for boolean values, `false` precedes `true`, so the order is specified as descending. That way the default machine (the one for which `true` is returned) appears first, and the rest of the machines (the ones for which `false` is returned) follow it in the ordering.

Within that ordering, the query orders the machines by the result of `overheadCost` (another function defined below the query).

The query selects the first machine in this overall order: the default machine if it is feasible, and the one with the lowest overhead otherwise.

- 4 Remove all the code following the two rules referred to by the query, resulting in the following code for the whole module:

```

machine = select first(m) from machines m _
      where MaxPartWidthCheck and MaxPartLengthCheck _
      order by isDefaultMachine(m) desc, overheadCost(m)



isDefaultMachine(m) = (m == defaultMachine)

//account for both types of machine/overhead cost in the system
overheadCost(m) = _
      ( m.workCenterLaborRate * m.workCenterOverheadMultiplier ) + _
      m.workCenterOverheadRate

Rule MaxPartWidthCheck: blank.serWidth <= m.bedWidth
Message MaxPartWidthCheck: m.name + _
      ' is not feasible. The blank width (' + roundBlankWidth + _
      ' mm) is greater than max bed width of the machine (' + _
      machineBedWidth + ' mm) '
roundBlankWidth= roundEps(blank.serWidth, 0.01)
machineBedWidth = m.bedWidth

Rule MaxPartLengthCheck: blank.serLength <= m.bedLength
Message MaxPartLengthCheck: m.name + _
      ' is not feasible. The blank length (' + roundBlankLength + _
      ' mm) is greater than max bed length of the machine (' + _
      machineBedLength + ' mm) '
roundBlankLength= roundEps(blank.serLength, 0.01)
machineBedLength = m.bedLength

```


- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).


Preferring One Class of Machines Over Another

Suppose that, for Abrasive Cut, you want to modify the machine selection to prefer on-site machines (as opposed to machines located at a remote facility). And suppose that (in order to support this) machines have a string-valued attribute, `isOnSite`, which is `true` if the machine is on site and `false` otherwise (see [Modifying a Machine Type—Preferring One Class of Machines Over Another](#)).

Follow these steps to modify machine selection for the AbrasiveJet Cut process so that it prefers customer-defined machines over predefined machines:

- 1 In the navigation pane, expand **Processes, GCDs & Operations** and double-click the AbrasiveJet Cut process.
- 2 In the editing pane, select the CSL tab.
- 3 Click on the folder icon, , next to `machineSelectionRule`.

The module text appears in the editing pane.

In this example, the module is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

- 4 Modify the query that appears in the module. The unmodified query is as follows (see [Removing unwanted machine checks](#)):

```
machine = select first(m) from machines m _
           where MaxPartWidthCheck and MaxPartLengthCheck _
           order by isDefaultMachine(m) desc, overheadCost(m)
```

To prefer all on-site machines over all off-site machines, modify the query so that it appears as follows:

```
machine = select first(m) from machines m _
           where MaxPartWidthCheck and MaxPartLengthCheck _
           order by isOnSite(m) desc,      overheadCost(m)

isOnSite(m) = (m.isOnSite == true)
```

This query orders the feasible machines by the result of `isOnSite` (a function defined below the query in this module). The result of `isOnSite` is either `true` or `false`. In the ascending order for boolean values, `false` precedes `true`, so the order is specified as descending. That way on-site machines (the ones for which `true` is returned) appear first, and the rest of the machines (the ones for which `false` is returned) follow them in the ordering.



Within that ordering, the query orders the machines by the result of `overheadCost` (another function defined below the query).

To prefer on-site machines to off-site machines with equal overhead, modify the query so that it appears as follows:

```
machine = select first(m) from machines m _
  where MaxPartWidthCheck and MaxPartLengthCheck _
  order by overheadCost(m), isOnSite(m) desc

isOnSite (m) = (m.isOnSite == true)
```

This query orders the feasible machines by overhead cost, and within that ordering, on-site machines appear first.

- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Adding Feasibility Rules

This section provides an example of adding feasibility rules to a process. The example uses the custom process, [AbrasiveJet Cut](#) (see [Adding a New Process to a Process Group](#)). Following [Waterjet Cut](#), this new process will include a feasibility module that performs the follow checks:

- Maximum material thickness check: blank thickness is less than or equal to the maximum blank thickness (as specified by a plant variable).
- Minimum material thickness check: blank thickness is greater than or equal to the minimum blank thickness (as specified by a plant variable).
- Maximum bundle thickness check: thickness of the stack of blanks to be processed is less than or equal to the maximum bundle thickness (as specified by a plant variable).

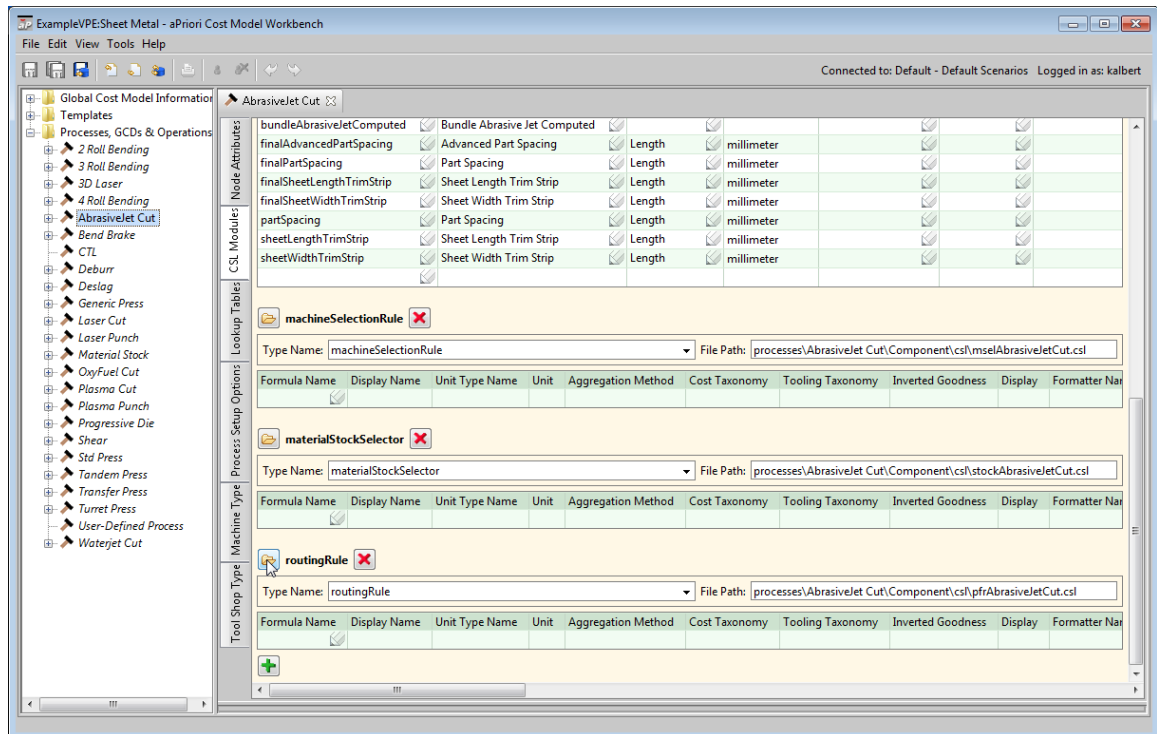
Note that a complete implementation of the [AbrasiveJet Cut](#) process must include operation-level feasibility rules as well as the process-level feasibility rules described here (see the operation-level feasibility modules for [Waterjet Cut](#)).

For more information on feasibility rules, see [Process and Operation Feasibility](#) in the chapter [Working with Cost Model Logic](#).


Follow these steps to define the process-level feasibility rules for [AbrasiveJet Cut](#):

- 1 In the navigation pane, expand **Processes, GCDs & Operations** and double-click the [AbrasiveJet Cut](#) process.
- 2 In the editing pane, select the CSL tab.

- 3 Click on the folder icon, , next to routingRule.



An empty feasibility module appears in the editing pane. This is a copy of Laser Cut's empty feasibility module.

In this example, the module is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

- 4 Enter the following code into the editing pane:

```
import libAbrasiveJetUtilities.csl

Rule MaxThicknessCheck: blank.thickness <=
plant.waterjetMaxPartThickness

Message MaxThicknessCheck: _
    'Material thickness is too thick for Waterjet Cutting.'

Rule MinThicknessCheck: blank.thickness >=
plant.waterjetMinPartThickness

Message MinThicknessCheck: _
    'Material thickness is too thin for Waterjet Cutting.'

Rule MaxBundleThicknessCheck: _
    GetBundleThickness <= plant.waterjetMaxPartThickness
```

```
Message MaxBundleThicknessCheck: _  
    'Part stack thickness is too thick for Waterjet Cutting.'
```

In the code above, `blank` is a CSL standard input that represents the blank from which the current part is created. The expression `blank.thickness` designates the thickness of a single blank. The expressions `plant.waterjetMaxPartThickness` and `plant.waterjetMinPartThickness` designate the values of the plant variables **waterjetMinPartThickness** and **waterjetMaxPartThickness**.

The formula `GetBundleThickness` must be defined in the library `libAbrasiveJetUtilities.csl`—see [Creating a New Library](#).

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Adding New Plant Variables

This section provides an example of adding new plant variables (also known as cost model variables) to a process group. The example uses the sample custom process `AbrasiveJet Cut`, which is copied from `Laser Cut`, and will mimic `Waterjet Cut` (see [Adding a New Process to a Process Group](#)). An additional example is also included, to support a further customization related to cycle time padding.

This section includes the following subsections:

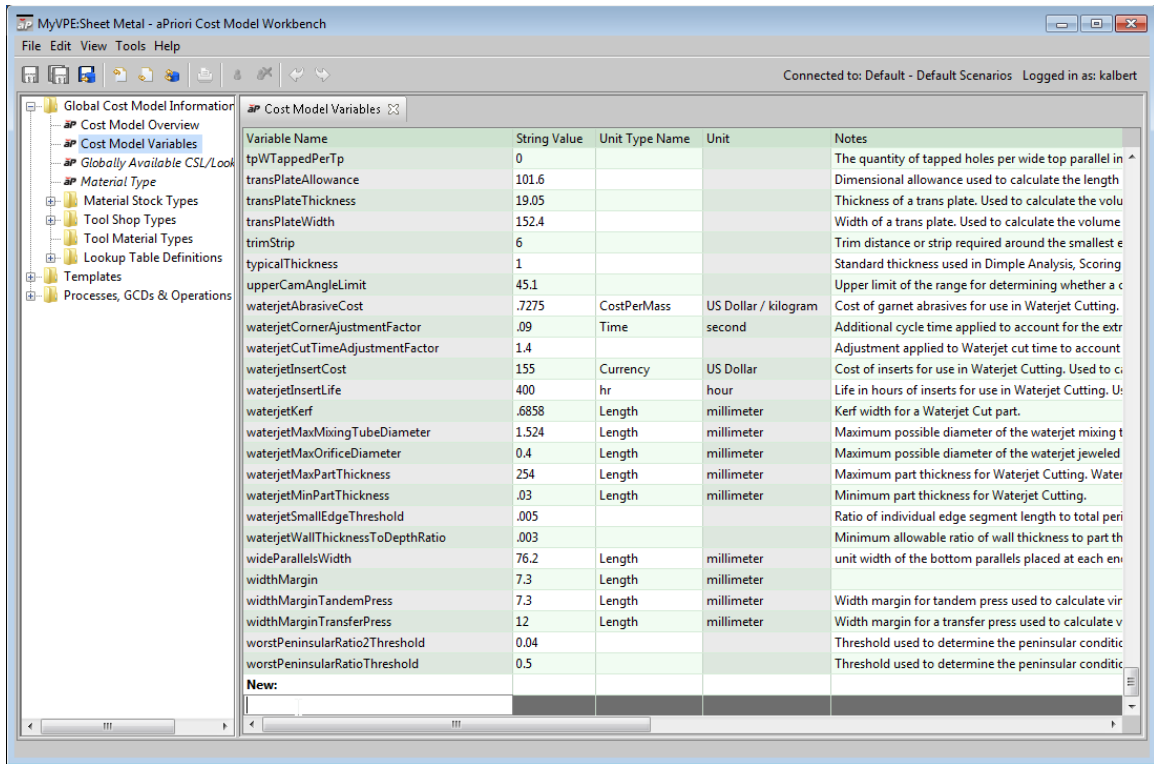
- [Adding Plant Variables for a New Process](#)
- [Adding Plant Variables—Padding Cycle Time](#)

For more information on plant variables, see [Working with Plant Variables](#) in the chapter [Working with Cost Model Data and Metadata](#).

Adding Plant Variables for a New Process

Follow these steps to add the `AbrasiveJet` plant variables:

- 1 In the navigation pane, expand **Global Cost Model Information**, and double click **Cost Model Variables**.
- 2 In the editing pane, scroll to the bottom, and double click the cell underneath **New**.



- Enter the information in the table below. Double-click a cell to enter a value. In the **Unit Type Name** field, when you select an item from the dropdown list, a value for the **Unit** field appears automatically.

Variable Name	String Value	Unit Type Name	Unit	Notes
abrasiveJetAbrasiveCost	.7275	CostPerMass	US Dollar / kilogram	Cost of garnet abrasives for use in AbrasiveJet Cutting. Used to calculate expendable tooling cost.
abrasiveJetCornerAdjustmentFactor	.09	Time	second	Additional cycle time applied to account for the extra time required to stop the feed rate at each corner during AbrasiveJet Cutting.
abrasiveJetCutTimeAdjustmentFactor	1.4			Adjustment applied to AbrasiveJet cut time to account for the extra time required for acceleration or deceleration to the required feed rate.
abrasiveJetInsertCost	155	Currency	US Dollar	Cost of inserts for use in AbrasiveJet Cutting. Used to calculate expendable tooling cost.

abrasiveJetInsertLife	400	hr	hour	Life in hours of inserts for use in AbrasiveJet Cutting. Used to calculate expendable tooling cost.
abrasiveJetKerf	.6858	Length	millimeter	Kerf width for an AbrasiveJet Cut part.
abrasiveJetMaxMixingTubeDiameter	1.524	Length	millimeter	Maximum possible diameter of the abrasiveJet mixing tube.
abrasiveJetMaxOrificeDiameter	0.4	Length	millimeter	Maximum possible diameter of the abrasiveJet jeweled orifice.
abrasiveJetMaxPartThickness	254	Length	millimeter	Maximum part thickness for AbrasiveJet Cutting. AbrasiveJets can actually cut thicker parts but the cutting model breaks down above this threshold.
abrasiveJetMinPartThickness	.03	Length	millimeter	Minimum part thickness for AbrasiveJet Cutting.
abrasiveJetSmallEdgeThreshold	.005			Ratio of individual edge segment length to total perimeter length below which an edge is considered small and must be cut using a slower feed rate.
abrasiveJetWallThicknessToDepthRatio	.003			Minimum allowable ratio of wall thickness to part thickness for an AbrasiveJet Cut part.

Adding Plant Variables—Padding Cycle Time

This section adds a plant variable to support cycle time padding. This variable can be used directly in a taxonomy module (see [Padding Cycle Time by Adding a Constant](#)), or to supply a default value for a process setup option (see [Adding a Setup Option—Padding Cycle Time](#)). See also [Modifying a Machine Type—Padding Cycle Time](#).

Follow these steps to define the variable:

- 1 In the navigation pane, expand **Global Cost Model Information**, and double click **Cost Model Variables**.
- 2 In the editing pane, scroll to the bottom, and double click the cell underneath **New:**.
- 3 Enter a new line in the table, specifying the following information:
 - o **Field Name:** AbrasiveJetCycleTimeAdditiveAdjustment

- **String Value:** 0
- **Unit Type Name:** Time

Adding New Process Setup Options

This section illustrates adding new process setup options to a process. The examples in this section use the custom process AbrasiveJet Cut (see Adding a New Process to a Process Group). The first two examples mimic setup options for the process Watercut Jet. The last example illustrates further customization. The section contains the following subsections:

- Adding a Setup Option—Using CSL, Formula, and User Modes
- Adding a Setup Option—Using List Mode to Access a Lookup Table
- Adding a Setup Option—Padding Cycle Time


For more information on process setup options, see Working with Process Setup Options in the chapter Working with Cost Model Data and Metadata.


Adding a Setup Option—Using CSL, Formula, and User Modes

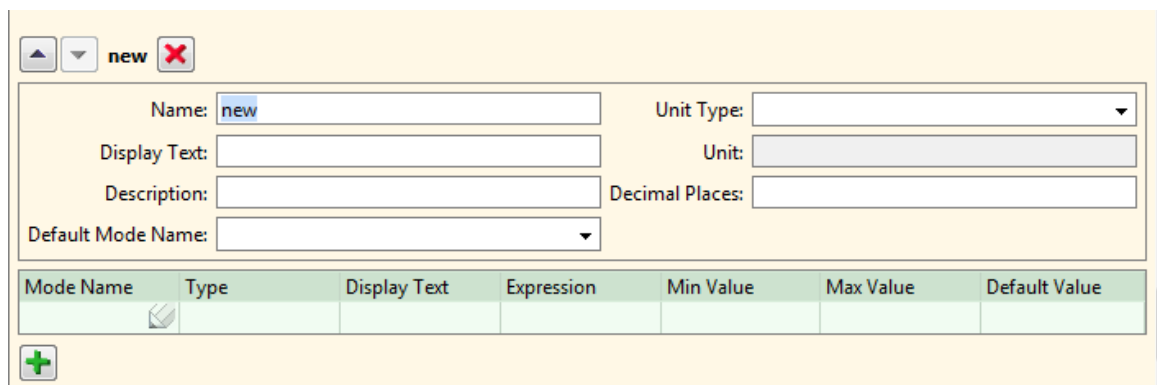
This section contains an example of adding a setup option to a process. Following Waterjet Cut, this new process will include a setup option, `bundleCount`, for the number of blanks that are stacked and processed simultaneously.

Follow these steps to create the setup option:

- 1 In the navigation pane, double-click **AbrasiveJet Cut**; in the editing pane, click the **Process Setup Options** tab.

In this example, the process is new, and so is editable without an explicit override. For inherited objects, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

- 2 Scroll to the bottom of the editing pane, and click . Text boxes and a table for a new setup option appear.



Each line of the table will define an end-user input mode. You will define three modes:


- Default mode: uses the default bundle count 1. This value is specified by a simple CSL expression, consisting of just a numerical literal. (See Adding a

Setup Option—Padding Cycle Time for another example of using CSL mode in a process setup option.)

- Computed mode: uses a computed bundle count. This value is specified by the CSL formula (defined in the process taxonomy module or a supporting library module) `bundleAbrasiveJetCountComputed`. (See `bundleWaterjetCountComputed` in the library module `libWaterjetUtilities.csl`.)
 - User override mode: This value is entered by the end user.
- 3** Add a line to the table, specifying the following column values:
- **Mode Name:** `aDefaultValue`
 - **Type:** `CSL`
 - **Display Text:** `Default Value`
 - **Expression:** `1`
- 4** Add a second line to the table, specifying the following column values:
- **Mode Name:** `aaComputedValue`
 - **Type:** `Formula`
 - **Display Text:** `Computed Value`
 - **Expression:** `bundleAbrasiveJetCountComputed`
- 5** Add a third line to the table, specifying the following column values:
- **Mode Name:** `aaaUserOverride`
 - **Type:** `User`
 - **Display Text:** `User Override`
 - **Min Value:** `1`
 - **Max Value:** `bundleAbrasiveJetCountComputed`
- 6** Enter the following information into the text boxes:
- **Name:** `bundleCount`
 - **Display Text:** `Number of Parts in Stack`
 - **Description:** `Parts can be stacked to cut multiples at once. Maximum number of parts is determined by part thickness and the maximum effective stack thickness without sacrificing part quality.`
 - **Default Mode Name:** `aDefaultValue`
 - **Decimal Places:** `1`

Mode Name	Type	Display Text	Expression	Min Value	Max Value	Default Value
aDefaultValue	CSL	Default Value	1			
bundleWaterjet...	FORMULA	Computed Value	bundleAbrasiveJetCountComputed			
aaaUserOverri...	USER	User Override	bundleAbrasiveJetCountComputed	1.0	bundleAbrasiveJetCountComputed	

7 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

8 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.


For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).


Adding a Setup Option—Using List Mode to Access a Lookup Table

This section contains an example of adding a setup option that allows the user to select from a list of values that are based on a lookup table. The example uses the custom process, AbrasiveJet Cut, which is copied from Laser Cut, and is almost identical to Waterjet Cut (see [Adding a New Process to a Process Group](#)). Following Waterjet Cut, this new process will include a setup option, `nozzleType`, that allows the user to specify a the type of nozzle in use for the current part. The nozzle characteristics affect part cost, and are contained in a lookup table (see [Adding Lookup Tables](#)). The lookup table is accessed by the process taxonomy module.

Follow these steps to create the setup option:

1 In the navigation pane, double-click **AbrasiveJet Cut**; in the editing pane, click the **Process Setup Options** tab.

In this example, the process is new, and so is editable without an explicit override. For inherited objects, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

2 Scroll to the bottom of the editing pane, and click . Text boxes and a table for a new setup option appear.

The table will define the end-user input mode, LIST mode, which provides a list of alternative values from which the user can choose. The alternatives are specified by a select expression (that is, a CSL query expression--see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#)) that queries a lookup table. There will be one alternative for each lookup table entry.

3 Add a line to the table, specifying the following column values:

- **Mode Name:** UserOverride
- **Type:** List
- **Display Text:** Nozzle Type


- **Expression:** `select k.nozzleType from tblAbrasiveJetAbrasiveNozzle k order by k.nozzleCost asc`
- **Default Value:** `'Mid-Life Composite Carbide'`

4 Enter the following information into the text boxes:

- **Name:** `nozzleType`
- **Default Mode Name:** `UserOverride`

Mode Name	Type	Display Text	Expression	Min Value	Max Value	Default Value
UserOverride	LIST	Nozzle Type	select k.nozzleType from tblAbrasiveJetAbrasiveNozzle k ...			Mid-Life Composite Carb...

5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

6 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.


For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).


Adding a Setup Option—Padding Cycle Time

This section contains an example of adding a setup option to a process. The example uses the sample custom process `AbrasiveJet Cut`, which is copied from `Laser Cut` and mimics `Waterjet Cut` (see [Adding a New Process to a Process Group](#)). Unlike `Waterjet Cut`, this new process will include a setup option, `cycleTimeAdditiveAdjustment`, for a value to be added to the process cycle time as an adjustment (see [Padding Cycle Time by Adding a Constant](#)). See also [Modifying a Machine Type—Padding Cycle Time](#) and [Adding Plant Variables—Padding Cycle Time](#).

Follow these steps to create the setup option:

1 In the navigation pane, double-click **AbrasiveJet Cut**; in the editing pane, click the **Process Setup Options** tab.

In this example, the process is new, and so is editable without an explicit override. For inherited objects, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

2 Scroll to the bottom of the editing pane, and click . Text boxes and a table for a new setup option appear.

Each line of the table will define an end-user input mode. You will define two modes:



- **Default mode:** uses the default bundle count specified by the plant variable `cycleTimeAdditiveAdjustment`. This value is specified by the CSL expression `plant.cycleTimeAdditiveAdjustment`.

- User override mode: This value is entered by the end user.
- 3 Add a line to the table, specifying the following column values:
 - **Mode Name:** aDefaultValue
 - **Type:** CSL
 - **Display Text:** Default Value
 - **Expression:** plant.cycleTimeAdditiveAdjustment
 - 4 Add a second line to the table, specifying the following column values:
 - **Mode Name:** aaUserOverride
 - **Type:** User
 - **Display Text:** User Override
 - **Default Value:** 0
 - 5 Enter the following information into the text boxes:
 - **Name:** cycTimeAdditiveAdjustment
 - **Display Text:** Cycle Time Additive Adjustment
 - **Description:** Value to be added onto cycle time as padding.
 - **Default Mode Name:** aDefaultValue
 - **Unit Type:** Time
 - **Decimal Places:** 1

▲ ▼ cycleTimeAdditiveAdjustment ✖

Name: <input type="text" value="cycleTimeAdditiveAdjustment"/>	Unit Type: <input type="text" value="Time"/>
Display Text: <input type="text" value="Cycle Time Additive Adjustment"/>	Unit: <input type="text" value="second"/>
Description: <input type="text" value="Value to be added onto cycle time as padding."/>	Decimal Places: <input type="text" value="1"/>
Default Mode Name: <input type="text" value="aDefaultValue"/>	

Mode Name	Type	Display Text	Expression	Min Value	Max Value	Default Value
aDefaultValue	CSL	Default Value	plant.cycleTimeAdditiveAdjustment			
aaUserOverride	USER	User Override				0

- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Padding Cycle Time by Adding a Constant](#) for an example of using this process setup option in a CSL module.

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Adding Lookup Tables

This section contains an example of adding a lookup table to a process. The example uses the custom process, AbrasiveJet Cut (see [Adding a New Process to a Process Group](#)). AbrasiveJet Cut (following Waterjet Cut) will use a lookup table, `nozzleType`, to store the cost and lifetime for various types of nozzles.

This section contains the following subsections:

- [Adding a Lookup Table Definition](#)
- [Adding a Lookup Table](#)




For more information on lookup tables, see [Working with Lookup Tables](#) in the chapter [Working with Cost Model Data and Metadata](#).

Adding a Lookup Table Definition



Before you create the new lookup table, you must create a new lookup table definition (the metadata for the lookup table). Follow these steps:

- 1 In the CMWB navigation pane, right-click and select **New > Lookup Table Definition**. The **New Lookup Table Definition** dialog appears.
- 2 Enter the `tblAbrasiveJetAbrasiveNozzle` in the **Name** field, and click **OK**. An empty definition table appears in the editing pane.
- 3 Add a line to the table, specifying the following column values (double click in a table cell to add a value):
 - **Field Name:** `nozzleCost`
 - **Unit Type Name:** `Currency`
 - **Property Type Name:** `double`
- 4 Add a second line to the table, specifying the following column values:
 - **Field Name:** `nozzleLife`
 - **Unit Type Name:** `hr` (Enter this value by typing, rather than by selecting from the dropdown list.)
 - **Property Type Name:** `double`
- 5 Add a third line to the table, specifying the following column values:
 - **Field Name:** `nozzleType`
 - **Property Type Name:** `string`

tblAbrasiveJetAbrasiveNozzle						
Name: tblAbrasiveJetAbrasiveNozzle						
Field Name	Description	Default Value Text	Unit Type Name	Unit	Property Type Name	Notes
nozzleCost			Currency	US Dollar	double	
nozzleLife			hr	hour	double	
nozzleType					string	

- Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- Click the  icon to the left of the table name in the editing pane. The column properties and column groups page appears in a new tab in the editing pane. Click **Nozzle Type** under displayed fields, and click the up arrow, , to move **Nozzle Type** to the top.

Available Fields:	Displayed Fields:
Lookup Table	Nozzle Type Nozzle Cost Nozzle Life
▶ ◀	
Column Group Controls + ×	
Column Properties	
Name:	<input type="text"/>
Display Name:	<input type="text"/>
Formatter:	<input type="text"/>
Parent Group:	<input type="text"/>


- Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.




Adding a Lookup Table

Now that you have a new lookup table definition, you can create the new lookup table. Follow these steps:

- In the CMWB navigation pane, double click the AbrasiveJet Cut process.



- 2 Click the **Lookup Tables** tab in the editing pane.

In this example, the process is new, and so is editable without an explicit override. For inherited objects, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.

- 3 Click  to add a new table. Fields for the new table appear in the editing pane.
- 4 Enter `tblAbrasiveJetAbrasiveNozzle` in the **Name** field.
- 5 Select `tblAbrasiveJetAbrasiveNozzle` in the **Meta Type** field.
- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes. The fields for the new table may be moved, so that they appear alphabetically with any other tables listed in the editing pane.
- 7 Click the folder icon, , above the **Name** field. The new, empty table appears in a new tab in the editing pane.
- 8 Rearrange the columns of the table by dragging the **Nozzle Type** column header all the way to the left-most column of the table.
- 9 Add a line to the table, specifying the following column values (double click in a table cell to add a value):
 - **Nozzle Type**: Premium Composite Carbide
 - **Nozzle Cost**: 150.00
 - **Nozzle Life**: 125.00
- 10 Add a second line to the table, specifying the following column values:
 - **Nozzle Type**: Mid-Life Composite Carbide
 - **Nozzle Cost**: 100.00
 - **Nozzle Life**: 85.00
- 11 Add a third line to the table, specifying the following column values:
 - **Nozzle Type**: Low-Cost Composite Carbide
 - **Nozzle Cost**: 50.00
 - **Nozzle Life**: 45.00

Nozzle Type	Nozzle Cost (USD)	Nozzle Life (hr)
Low-Cost Composite Carbide	50.00	45.00
Premium Composite Carbide	150.00	125.00
Mid-Life Composite Carbide	100.00	85.00

- 12 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

13 To incorporate your change  to the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

See [Adding a Setup Option—Using List Mode to Access a Lookup Table](#) for an example of basing a process setup option on the entries in this table.

Modifying Taxonomy Modules

This section contains an example of modifying a process taxonomy module. The example uses the custom process AbrasiveJet Cut (see [Adding a New Process to a Process Group](#)), and focuses on the high-level cycle time formulas. The first two sections sketch how to modify the AbrasiveJet Cut (which is copied from Laser Cut) so that its cycle time calculations mimic those of Waterjet Cut. The last section illustrates a further customization.

This section includes the following subsections:

- [Modifying the Formula Table](#)
- [Modifying the Cycle Time Formulas](#)
- [Padding Cycle Time by Adding a Constant](#)

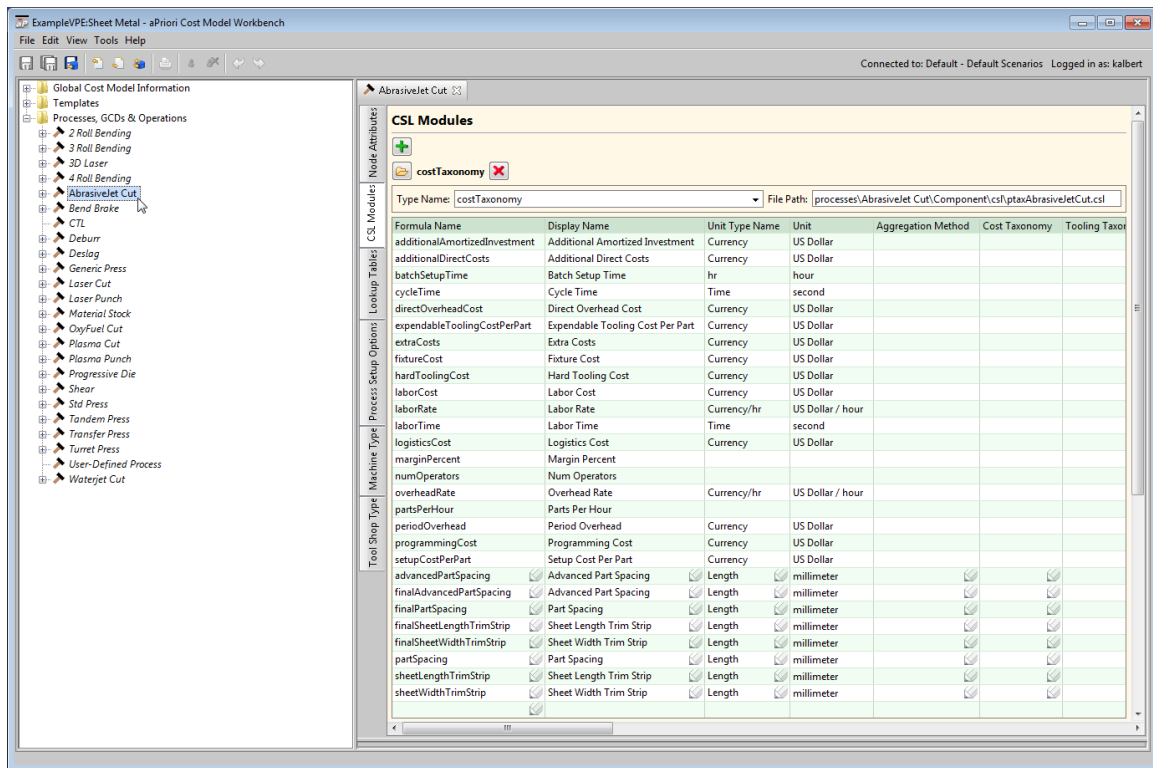
For more information on taxonomy modules, see [Process and Operation Taxonomy](#) in the chapter [Working with Cost Model Logic](#).


Modifying the Formula Table

If you modify a taxonomy module to include new output formulas, you must modify its associated formula table. This section shows you how to modify the AbrasiveJet Cut formula table, copied from Laser Cut, so that it mimics Waterjet Cut.

Follow these steps to navigate to the formula table:

- 1** In the navigation pane, double-click **AbrasiveJet Cut**.
- 2** In the editing pane, click the **CSL Modules** tab. The formula table for the process taxonomy file (named **costTaxonomy**) appears near the top of the editing pane.



In this example, the process is new, and so is editable without an explicit override. To modify inherited nodes, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.


The process taxonomy module for AbrasiveJet Cut (following Waterjet Cut) has all the output formulas that Laser Cut has, plus the following output formulas:



- bundleAbrasiveJetComputed
- bundleAbrasiveJetCountComputed
- defaultAbrasiveFlowRate
- defaultFeedRateLargeFeatures
- defaultFeedRateSmallFeatures
- defaultLargeFeatureFeedRate
- defaultMixingTubeDiam
- defaultOrificeDiam
- defaultSmallFeatureFeedRate
- defaultOrificeDiam
- rapidTraverseTime

(Note that if you've already defined a process setup option that refers to a formula, the formula is automatically added to the formula table.)

For each formula listed above, follow these steps:

- 3 Click in the empty Formula Name field of the last line of the table.

- 4 Enter information in the following fields:
- **Formula Name:** enter name of the formula (one of the names listed above)
 - **Display Name:** enter the name as you want it to appear in end user tables and reports.
 - **Unit Type Name:** select the type of units for the formula value.
 - **Unit:** units for the formula value. This field is not editable; it is determined by the Unit Type Name field.
 - **Inverted Goodness:** set this to false. (This toggles the red/green arrows in the UI. By default ("false"), smaller values are considered desirable (as in the case of costs), and therefore are displayed with green arrows. However, a smaller value for an item such as feed rate could be considered a negative, and setting **Inverted Goodness** to "true" allows it to be displayed with a red arrow.)
 - **Display:** set this to true if you want the formula value included in end user tables and reports; set this to false otherwise.
 - **Dependency Tree Visibility:** Set this value to determine the behavior of the right-click **Show Formula Dependencies** command in the UI, so that it displays information that is relevant to the end user. Possible settings are ALWAYS, NEVER, WHEN_NONZERO, or unset. Currently the **Formula Dependencies** window is populated only for direct and indirect rates. The "correct" setting for a given formula is somewhat subjective, but here are some guidelines:
 - WHEN_NONZERO – Use this if formula relevance is determined by another setting such as a cost model variable or a site variable. To have formula relevance be driven by these global toggles, you need to set the visibility to WHEN_NONZERO and (when it is irrelevant in the calculation) force it to evaluate to zero.
 - NEVER – You would typically use this setting only when you have a formula that does not show up in the UI, but which could be referenced in a spreadsheet report. Or for a formula that will be used in PSOs and which is collected by a parent node. Other possible uses would be for a standard formula whose equations are not set up properly to display in the dialog, or for an insignificant calculation such as a "fudge factor".
 - ALWAYS – This is the most typical selection for new formulas. The formula will show up in the dependency tree and the user will be able to see (and override) the value directly from the Formula Dependencies window even if it computes to zero.
- Note:** If **Dependency Tree Visibility** is unset, the behavior is the same as if it had been set to ALWAYS.
- **Overridable:** set this to true if users should be able to override the value for this formula in the UI.
- Note:** At the Site Cost Model level, this column is labeled **Overridable At**, and its values define contexts in which the result of the particular formula can be overridden: BRANCH, PROCESS, OPERATION, or UTILIZATION_PROCESS.
- **Description:** enter an optional description of the formula.
- 5 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

- 6 To incorporate your change  to the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).



Modifying the Cycle Time Formulas

This section contains an example of modifying cycle time formulas for AbrasiveJet Cut, which is copied from Laser Cut. The calculations for AbrasiveJet cut (following Waterjet Cut) differ from Laser Cut in the following major areas:

- Cycle time
- Feed rate
- Pierce time
- Expendable tooling costs

This example focuses on the high-level cycle time formulas for the AbrasiveJet Cut process taxonomy module. To complete the modification of the process (copied from Laser Cut), you must also modify formulas associated with expendable tooling in the process taxonomy module, as well as formulas associated with feed rate and pierce time in library modules. In addition, you must change the content of the child operation modules as sketched at the end of step 7, below. (The Waterjet Cut process is fully implemented along these lines.)

Follow these steps to modify the high-level cycle time formulas in the process taxonomy module:

- 1 In the navigation pane, double-click **AbrasiveJet Cut**; in the editing pane, click the **CSL Modules** tab.
- 2 Click  next to **cost Taxonomy**. The CSL module appears in the editing pane.
In this example, the module is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 3 In the editing pane, remove the import of the library `libContourCutting.csl`.
This library contains calculations that are shared among processes such as Laser Cut, Plasma Cut, and OxyFuel Cut. These calculations relate to cycle time, traverse time, feed rates, and pierce time. AbrasiveJet Cut (following Waterjet Cut) does not use this library, but instead uses its own formulas for these calculations.
- 4 After the other import lines, add the following:

```
import libAbrasiveJetUtilities.csl
```

This library will contain utilities that contribute to the calculation of feed rates, pierce times, bundle sizes, and expendable tooling costs. See [Creating a New Library](#), below.

5 Replace the cycle time formula

```
cycleTime = GetCycleTime_SheetMetal_LaserCut(processTime,
numOperators)
```

with

```
cycleTime = GetCycleTime_SheetMetal_AbrasiveJetCut(processTime,
numOperators)
```

The function `GetCycleTime_SheetMetal_AbrasiveJetCut` must be defined in `libAccounting_SheetMetal.csl`. See [Modifying a Library Module](#), below. In this and a related library, `cycleTime` is defined as the product of `processTime` and a plant-variable-specified adjustment factor (a global adjustment that applies across all processes in the process group). The value `num_operators` is not used in this model; it is present only for consistency with the library functions for other sheet metal processes.

6 At the end of the module replace the following formula:

```
processTime = (sumOpsCycleTime + processCycleTime) /
machine.numHeads
```

with this formula:

```
processTime = ((sumOpsCycleTime + processCycleTime) / _
GetBundleAbrasiveJetCount) / machine.numNozzles
```

The function `GetBundleAbrasiveJetCount` should be defined in `libAbrasiveJetUtilities.csl`. It determines the number of blanks that are stacked and processed simultaneously.

7 Add these lines to the end of the module:

```
sumOpsCycleTime = select sum(ops.cycleTime) from childOps ops //
secs / part
processCycleTime = rapidTraverseTime
```

```
rapidTraverseTime = numOps * (averageFeatureDistance / _
machine.rapidTraverseRate) * SEC_PER_MIN
```

```
numOps = select count(*) from childOps op
```

```

numHoles = select count(op) from childOps op _
           where isSimpleHole(op) or isComplexHole(op)

averageFeatureDistance = { _
  (((part.blankBoxLength + part.blankBoxWidth) / 2) / (numHoles
/ 2)) _
  if (numHoles > 0) _
  (part.blankBoxLength + part.blankBoxWidth) / 2) otherwise _
}

defaultFeedRateLargeFeatures = GetFeedRateLargeFeatures
defaultFeedRateSmallFeatures = GetFeedRateSmallFeatures



```

The following formulas are defined in `libAbrasiveJetUtilities.csl`:

- o `GetFeedRateLargeFeatures`
- o `GetFeedRateSmallFeatures`

The feed rates are calculated here only to supply default-mode values to the feed rate process setup options.

Feed rates are used in the child operations to calculate cut time for each edge (of the blank or of a complex hole) and for each simple hole. Cut times are added to pierce time (for the blank and each hole), to yield cycle times for child GCDs of the part. These cycle times are summed in the formula `sumOpsCycleTimes`, above.

- 8 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 9 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.



For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Padding Cycle Time by Adding a Constant

AbrasiveJet Cut, as with Waterjet Cut and various other starting point processes, provides an example of adjusting cycle time through multiplication by a plant-variable-specified constant (see [Modifying the Cycle Time Formulas](#)). This is a global adjustment factor that applies across all processes in the process group. Another form of adjustment is the process-specific addition of (rather than multiplication by) a custom constant. Custom constants can be supplied by plant variables (see [Adding Plant Variables—Padding Cycle Time](#)), process setup options (see [Adding a Setup Option—Padding Cycle Time](#)), or machine attributes (see [Modifying a Machine Type—Padding Cycle Time](#)).

To pad AbrasiveCut's cycle time in this way, follow these steps:

- 1 In the navigation pane, double-click **AbrasiveJet Cut**; in the editing pane, click the **CSL Modules** tab.

- 2 click  next to **cost Taxonomy**. The CSL module appears in the editing pane.
In this example, the module is new, and so is editable without an explicit override. For inherited modules, you must select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 3 In the editing pane, find the `cycleTime` formula. Here is the formula for AbrasiveJet Cut (see [Modifying the Cycle Time Formulas](#)):

```
cycleTime = _
    GetCycleTime_SheetMetal_AbrasiveJetCut(processTime,
numOperators)
```

Supply the adjustment constant as a plant variable if the adjustment is the same regardless of the machine selected (see [Adding Plant Variables—Padding Cycle Time](#)). In this case, change the formula to this:

```
cycleTime = _
    GetCycleTime_SheetMetal_AbrasiveJetCut(processTime,
numOperators) + _
        plant.AbrasiveJetCycleTimeAdditiveAdjustment
```



Supply the adjustment constant as a machine attribute to allow it vary according to the selected machine (see [Modifying a Machine Type—Padding Cycle Time](#)). In this case, change the formula to this:

```
cycleTime = _
    GetCycleTime_SheetMetal_AbrasiveJetCut(processTime,
numOperators) + _
        machine.cycleTimeAdditiveAdjustment
```

Supply the adjustment constant as a process setup option if you want to allow the end user to customize the adjustment (see [Adding a Setup Option—Padding Cycle Time](#)). In this case, change the formula to this:

```
cycleTime = _
    GetCycleTime_SheetMetal_AbrasiveJetCut(processTime,
numOperators) + _
        setup.cycleTimeAdditiveAdjustment
```

- 4 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.

- 5 To incorporate your change  to the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Adding and Modifying Library Modules

This section contains examples of adding and modifying CSL library modules as part of the implementation of the custom process AbrasiveJet Cut (see [Adding a New Process to a Process Group](#)). The first example creates a library that is specific to AbrasiveJet Cut, and contains utilities that contribute to the calculation of feed rates, pierce times, bundle sizes, and expendable tooling costs. The second example adds a cycle-time related formula to the library `libAccounting_SheetMetal.csl`. This library is shared across processes in the sheet metal process group. Both these libraries are imported by the AbrasiveJet Cut process taxonomy file (see [Modifying the Cycle Time Formulas](#)).





This section includes the following subsections:

- [Creating a New Library](#)
- [Modifying a Library Module](#)



For more information on CSL, see [CSL Language Overview](#) in the chapter [Working with Cost Model Logic](#).

Creating a New Library

Follow these steps to create the new library for AbrasiveJet Cut:



- 1 In the navigation pane, expand **Global Cost Model Information**, and click **Globally Available CSL/Lookup Tables**.
- 2 In the editing pane, click **Library CSL**.
- 3 Select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 4 Click  near the top or bottom of the editing pane. Information fields for the new library appear at bottom of the editing pane.
- 5 Scroll to the bottom of the editing pane, and replace **new** with `libAbrasiveJetUtilities.csl` in the **File Name** field.
- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 Scroll to the top of the editing pane, and click  next to the name of the new library. An empty CSL module appears in the editing pane.
- 8 Enter the contents of the new library. In real development, you would create the contents from scratch. For the current example, since AbrasiveJet Cut is almost identical to Waterjet Cut, you can copy the contents of `libWaterjetUtilities.csl`,

and replace `Waterjet` with `AbrasiveJet` and `waterjet` with `abrasiveJet`, throughout.

- 9 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 10 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.

Modifying a Library Module

Follow the steps below to add an AbrasiveJet Cut cycle time formula to the library `libAccounting_SheetMetal.csl`. A complete implementation of AbrasiveJet Cut would include similar modifications for several other quantities, such as labor time, labor cost, and direct overhead cost.

- 1 In the navigation pane, expand **Global Cost Model Information**, and click **Globally Available CSL/Lookup Tables**.
- 2 In the editing pane, click **Library CSL**.
- 3 Select **Override Object** from the **Edit** menu, or click the override icon, , in the toolbar.
- 4 In the editing pane, click  next to the name of the new library. The CSL module appears in the editing pane.
- 5 Find the group of formulas whose names start with `GetCycleTime_SheetMetal_` (type control-f to search, F3 for next, and Shift+F3 for previous). Add the following formula at the end of the group:

```
GetCycleTime_SheetMetal_WaterjetCut(processTime, numOperators) =
CycleTime0
```



Note that `CycleTime0` is defined in the same library as follows:

```
CycleTime0 = GetCycleTime(processTime, numOperators)
```

The function `GetCycleTime` is defined in `libCommonAccounting.csl` in terms of `processTime`:

```
GetCycleTime(processTime, numOperators) = _
    processTime * plant.cycleTimeAdjustmentFactor
```

The formula `processTime` is defined in the taxonomy file proper. See [Modifying the Cycle Time Formulas](#).

- 6 Select **Save** from the **File** menu, or click  in the toolbar, to save your changes.
- 7 To incorporate your changes into the cost model, select **Publish Cost Model and VPE** from the **File** menu, or click  in the toolbar.



www.apriori.com

 **aPriori**

aPriori Technologies, Inc.

300 Baker Avenue

Concord, MA 01742

www.apriori.com