Best Practices for Options and Variants
September 2007

Introduction 2
- Product Development Approaches 2
- Windchill 9.0 Options and Variants for CTO and ETO Needs 3

Naming Conventions for Parameters and Constraints 3

Understanding How Logic Elements Are Evaluated 4

Selecting Options 6
- Enabling a Single Option Based on a Yes or No Response 6
- Enabling a Single Option Based on a User Response 6
- Choosing Between Two Options Based on a User Response 7
- Selecting One Option from a Set of Several Possible Options 8
- Selecting One Option from a Set of Many Known Options 10
- Selecting a Single Option from a Rapidly Changing Set of Options 12

Common Logic Expressions 12
- Comparing Two String Values 12
- Combining Two Expressions with an AND 13
- Including Two Expressions with an OR 14
- Making Two Non-String Expressions Equal 15
- Simple If/Then Expressions using Case Tables 15
- Complex If/Then Expressions using Case Tables 16
- Controlling Available Options using Case Tables 17
- Conditional Expressions 18

Creating Modular Generic Product Structures 18
- Designing a Module Product Structure 18
- Passing Parameter Values between Sections of a Product Structure Using Equivalencies . 22
- Passing Parameter Values between Sections of a Product Structure Using Reference IDs 23

Improving the User Interface of the Configuration Process 25
- Hiding Input Parameters When Users Do Not Have a Choice 25
- Explaining Input Parameters that Users May Not Specify 26
- Adding Images with Parameters with Enumerated Values 28

Displaying Values as Radio Buttons....30
Controlling Input Parameters Using Page Breaks 31
Adding Titles to Specific Pages of Parameters....33
Adding Images for Specific Pages of Parameters 34
Controlling the Order of Input Parameters Using Child Resolution....36
Using Supporting Documents in a Generic Product Structure 37
Attaching Documents to a Generic Product Structure 37
Releasing a Generic Product Structure....38
Defining a Default Baseline 38
Modifying a Default Baseline 38
Using a Default Baseline 39

Introduction

Product Development Approaches

In order for a company to successfully sell a product to a wide range of customers with a variety of needs, the product is often designed with the flexibility to offer a range of key capabilities. There are several common product development approaches that companies use to develop and sell flexible products. While definitions for these approaches vary by specific industry, these approaches can roughly be defined as the following:

Assemble-to-Order (ATO)

ATO is an approach to design products with a finite list of discrete option choices for key product features. Once the product design has been completed, the processes of customer ordering and manufacturing of the product are executed without involvement from the product development team. This approach is commonly used in many industries such as passenger vehicles, large and heavy equipment, computers, as well as other products manufactured in large volumes.

This approach is sometime known as Build-to-Order.

Configure-to-Order (CTO)

CTO is an approach to design flexible products that can be configured or customized to fit the unique needs of each customer order. Typically, a CTO product order is created with a configuration that applies rules and variable customer requirements to create a unique version of the product. The product development team defines the general product and often maintains the configuration logic for creating orders. The product development team typically has no involvement in processing or approving each specific customer order. This technique is common in industries such as industrial products, as well as automotive and aerospace suppliers.

Engineer-to-Order (ETO)

ETO is an approach similar to Configure-to-Order in that it involves fitting a general product to unique customer requirements. Additionally, an ETO process requires some involvement from the product development team before the product design is complete. Typically, an ETO process involves the product development team to define and validate each product configuration before manufacturing. As with the CTO process, the ETO approach is also common in industries such as industrial products, as well as automotive and aerospace suppliers.

Assemble-to-Stock

Assemble-to-Stock is an approach to design a general product with several discrete variations with small changes in features that address individual market or sales channel needs. This approach is commonly used in the consumer products industries where a product may be mass produced in several colors, sizes, or varying levels of key features.

Windchill 9.0 Options and Variants for CTO and ETO Needs

Product development teams have a range of needs to streamline the use of these product development approaches in their businesses. The Windchill 9.0 Options and Variants capabilities have been designed to help companies streamline their Configure-to-Order and Engineer-to-Order product development needs. These new Options and Variants capabilities are particularly valuable for organizations that need to incorporate a configuration to generate specific customer variant orders that are manufactured or refined by the product development team. Additionally, PTC is working on capabilities within Windchill to help streamline and optimize other product development approaches such as Assemble-to-Order in future Windchill releases.

The information within this document is provided to help companies plan and use Windchill 9.0 Options and Variants for CTO and ETO product development.

Naming Conventions for Parameters and Constraints

When you create parameter and constraint logic elements, establish a naming convention to facilitate working with, and understanding, structures of generic parts. A naming convention is especially important if multiple users work with these capabilities or if you expect to maintain a generic part structure for an extensive period of time.

Consider the following examples:

Logic Element	Usage	Naming Convention	Examples
Input Parameter	Obtain information from the user.	Preface an appropriate name with **ask**.	askName askOption

Logic Element	Usage	Naming Convention	Examples
Boolean Parameter	Applied on the usage link between a generic part and a child part to enable or disable the usage of the child part. In the Product Structure Explorer, the usage link is the **Inclusion Option** field displayed in the **Uses** tab.	Preface an appropriate name with **use**.	useOption1 useOption2
Case Table Constraint	Selects a value or set of values based upon a set of inputs.	Preface an appropriate name with **pick**.	pickColor pickSize
Constraint	Sets a parameter to a particular value based upon a condition or conditions.	Preface an appropriate name with **set**.	setColor setName
Reference Constraint	Establishes a reference between two parameters, which results in the value of one parameter being duplicated to the other parameter.	Preface an appropriate name with **ref**.	refSize refWeight

Understanding How Logic Elements Are Evaluated

Having a fundamental knowledge of how the system evaluates logic elements is extremely useful when you design and implement a generic product structure. This section provides an overview of the evaluation process.

The logic elements used by the system are divided into two fundamental categories – constraints and parameters. The system processes the logic elements for a particular product structure using the following procedure:

1. The parameters and constraints for the product structure are loaded into the system from the generic parts of the structure.
2. The system determines the input parameters that need to be displayed to the user in the Specification Editor. By default, all input parameters for the top-most generic part are processed first.
   - If at least one page break has been defined for the top-most generic part, then only the input parameters for the first page of this generic part are processed.
   - If the top-most generic part does not contain any input parameters, the system automatically selects another generic part by examining the logic of the product structure and processes its input parameters; however, if the selected generic part has a page break defined, then only those input parameters for the first page are processed.
   - If a child resolution has been defined, the system processes the input parameters for the identified child generic part. The processing of

input parameters for the child generic part also respects any child resolution or page breaks defined on the child generic part.

3. Any constraints that are applicable to the identified input parameters are applied, which may reduce or eliminate the values that are permissible for each input parameter.

   For example, if you have a parameter that includes 1,2,3,4, and 5 as valid values and a case table that only allows values of 1,2,3 or 4 for the same parameter – the parameter's list of permissible values would be reduced to only include 1,2,3 or 4.

4. The identified input parameters are displayed in the Specification Editor, including:

   – Images that are relevant to a parameter

   – Page titles that have been defined

   – Images that are relevant to the current page

   – Custom help pages that are relevant to the current page

5. After you select **Apply** or **Next**, the system processes all values on the current page. If you select:

   – **Apply** - the current page is displayed again.

   – **Next** - the next page is displayed.

   Note: The order of the parameters on a particular input page is not relevant because the system processes input parameters on a page-by-page basis, not on a parameter-by-parameter basis. Therefore, you should arrange the parameters on an input page in an order most likely to be clearly understood by those who are going to be configuring this product structure with the Specification Editor.

6. In some cases, the system automatically skips one or more pages of inapplicable input parameters based upon the values you entered or selected.

   For example, consider a product structure of generic parts that includes two parameters, *P1* and *P2*, with a page break in-between so that *P1* is displayed on Page 1 and *P2* would is on Page 2.

   If a case table is also defined so that if *P1*= **5**, then *P2* could only be **3** and the UI Property *hide when driven* for *P2* had been set to **true**.

   If you then selected **5** for *P1*, then Page 2, and *P2,* would be skipped because the value for *P2* has been automatically set to **3** and the system was told to skip (or hide) this parameter if its value had been driven (or set).

7. Once the system has identified values for all of the required input parameters, the system displays the **Input Review** page in the Specification Editor where you can review all of the input parameters that you specified and navigate to the **Solutions Page**.

Selecting Options

This section covers selecting options for a generic part from one or more possible candidates.

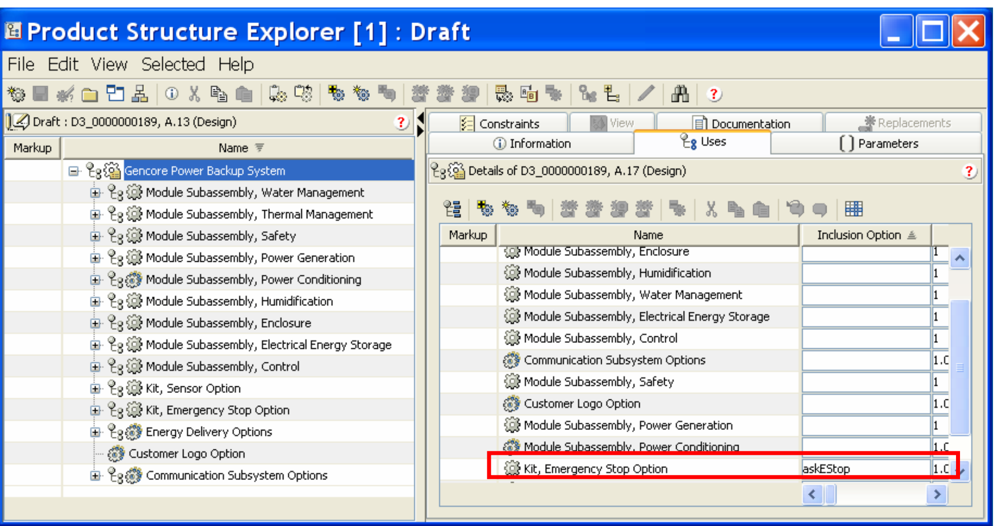
Enabling a Single Option Based on a Yes or No Response

In many situations, you may want to enable or disable a single option based upon a user response to a yes/no question. For example, you may wish to determine whether or not the user wants a particular option package.

In this case, the suggested approach is:

1. Create a generic part.
2. Attach a child part that represents the relevant option.
3. Add a Boolean input parameter to the generic part.
4. Define an appropriate prompt value for this parameter.
5. Add this Boolean parameter to the **Inclusion Option** of this generic part for the child part.

The following image shows an example of this technique where the **Emergency Stop Option** is or is not included by the value of the Boolean parameter askEStop.



Enabling a Single Option Based on a User Response

In some cases, you may want to enable or disable a single option based upon a user response which is not a yes/no question. For example, you may want to determine if the user wants the standard or deluxe package and to enable an additional subsystem if the user selected the deluxe package.

In this case, the suggested approach is:

1. Create a generic part.

2. Attach a child part that represents the relevant option.
3. Add a string input parameter to the generic part (For example, askPackage)
   - Define an appropriate prompt value for the string parameter.
   - Define a set of constraints for the string parameter, such as:
     - Deluxe
     - Standard
4. Add a Boolean non-input parameter to the generic part (For example, useDeluxe)
   - Define the default expression as follows:

     ```
     askPackage.equals("Deluxe")
     ```
5. Set the Boolean parameter useDeluxe on the inclusion option of this generic part for the child part of this option.

Using this approach, if the user's response is **Deluxe**, then the expression for the Boolean parameter useDeluxe evaluates to **true** and the child part is included in the variant part structure.

Choosing Between Two Options Based on a User Response

One typical situation occurs when the user's response is used to select one of two available options. For example, you might want to select the standard duty battery or the extended duty battery for a particular product.

In this case, the suggested approach is:

1. Create a generic part.
2. Attach two child parts, one for each of the relevant options.
3. Add a Boolean input parameter to the generic part; for example, askExtendedDuty.
   - Define an appropriate prompt value for this parameter, such as: *Do you want the extended duty battery?*
4. Add this Boolean parameter to the **Inclusion Option** of the generic part for the child part that corresponds to the extended duty battery.
5. Add a second Boolean non-input parameter, such as useStandardDuty, to the generic part
   - Define the default expression for this parameter as follows: `!askExtendedDuty`
6. Add the second Boolean parameter useStandardDuty to the Inclusion Option of this generic part for the child part that corresponds to the standard duty battery.

Using this approach, the user is presented with a single question to determine whether they want the extended duty battery or not. The user's

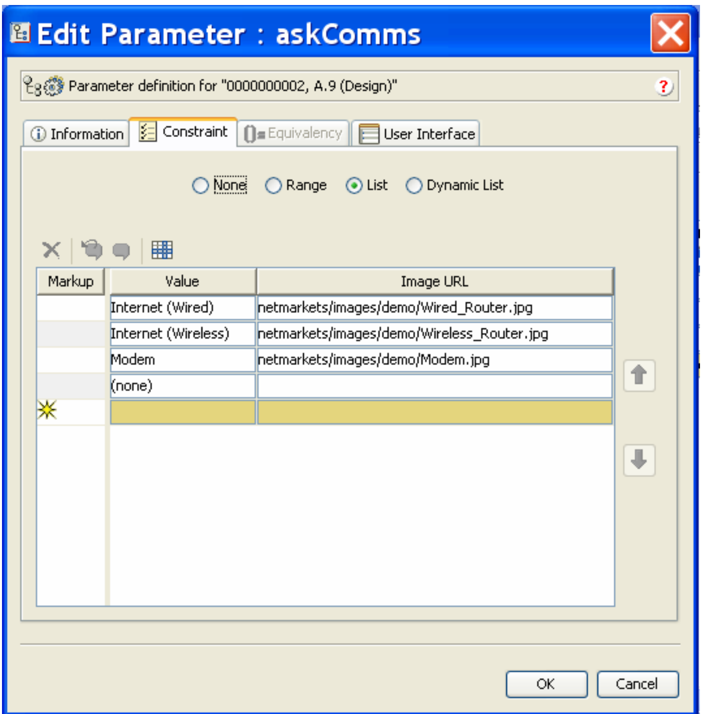
response is automatically used to include or not include the extended duty battery and the opposite of the user's response is used to exclude or not exclude the standard duty battery.

Selecting One Option from a Set of Several Possible Options

You may want to select a single option from a set of several possible options, based upon the user's response. For example, a remote power generation system could be available with three different communications and monitoring systems, such as modem, wired internet, and wireless internet, and you want to enable the correct system based upon the user's response.

In this case, the suggested approach is:

1. Create a generic part.
2. Attach a child part for each of the relevant finishing options.
3. Add a string input parameter to the generic part; for example, askComms.
   - Define an appropriate prompt value for the string parameter.
   - Define a set of constraints for the string parameter, such as:
     - **Internet (Wireless)**
     - **Internet (Wired)**
     - **Modem**

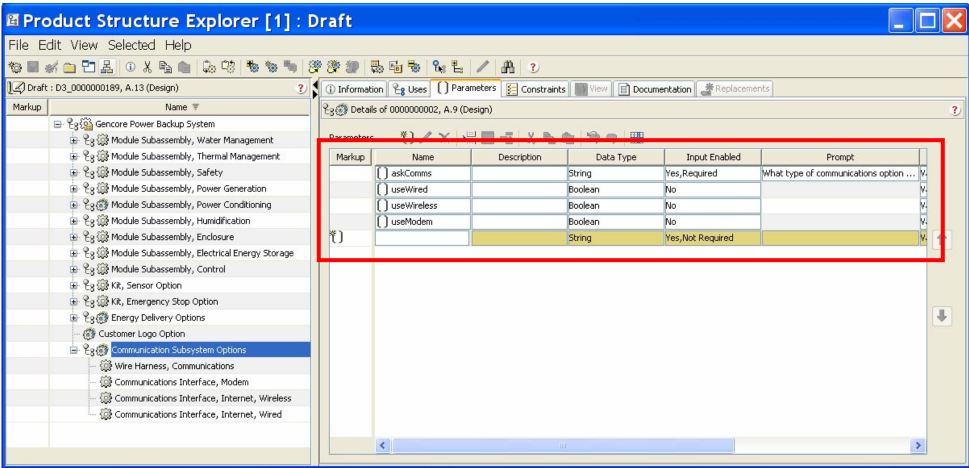


4. Add three Boolean non-input parameters to the generic part, such as:

   – useWireless – to represent the usage of the **Internet (Wireless)** system

   – useWired – to represent the usage of the **Internet (Wired)** system

   – useModem – to represent the usage of the **Modem** system



5. Add the Boolean parameters to the **Inclusion Option** of this generic part for the relevant child part for each option, as follows:

   – useWireless is defined for the **Inclusion Option** for the Wireless Internet sub-assembly.

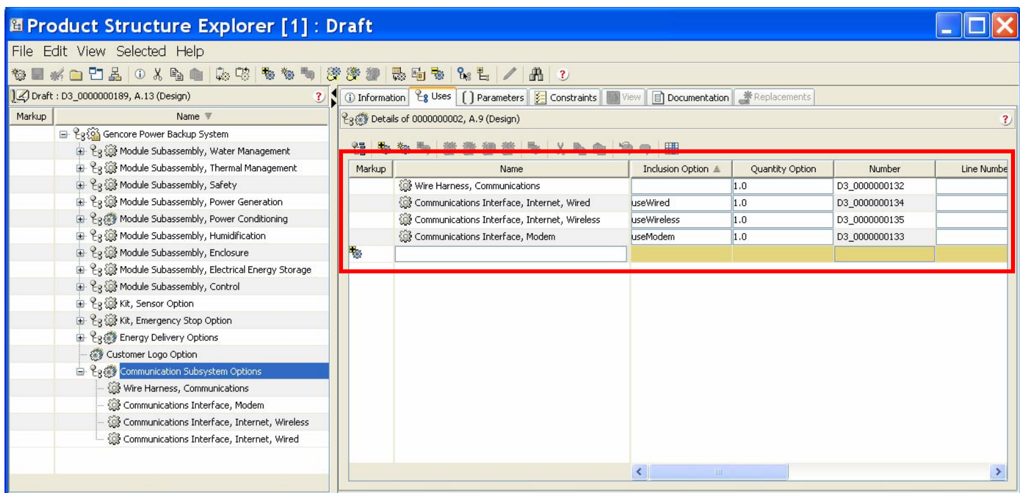
   – useWired is defined for the **Inclusion Option** for the Wired Internet sub-assembly.

   – useModem is defined for the **Inclusion Option** for the Modem sub-assembly.

6. Establish a case table constraint to map the user's responses to the correct Boolean parameter values. For example, consider the case table constraint pickComm.

askComms	**useWireless**	**useWired**	**useModem**
Internet (Wireless)	yes	no	no
Internet (Wired)	no	yes	no
Modem	no	no	yes

In this example, the user's response is captured in the parameter askComms which is constrained to three possible values. Based upon the user's response, one of the Boolean parameters is set to **true** by the case table constraint, which enables the corresponding sub-assembly.



Selecting One Option from a Set of Many Known Options

You may wish to select a single option from a set of several possible options based upon the user's response; however, there may be so many options that a single case table with a column allocated for each option may be difficult to organize or maintain.

For example, a wooden table top might be available in a very large number of materials and finishes such as Oak Veneer, Solid Oak, Solid Oak with Maple Trim, Maple Veneer, Solid Maple, Solid Maple with Oak Trim, Cherry Veneer, Solid Cherry, and Solid Cherry with Maple Trim.

To make all of this information more manageable, the goal is to introduce another parameter to connect the case table with the Boolean parameters that determine which optional sub-assembly is included. In this manner, the user's request is captured in an input parameter that is evaluated by a case table of valid options. The results, or output, of the case table is captured in a single parameter. Finally, the value for each Boolean is determined by evaluating this intermediary parameter.

In this case, the suggested approach is:

1. Create a generic part.
2. Attach a child part for each of the relevant finishing options.
3. Add a string input parameter to the generic part; for example, askFinish.
   - Define an appropriate prompt value for the string parameter.
   - Define a **List Constraints** for the string parameter with these values:
     - Oak Veneer
     - Solid Oak
     - Solid Oak with Maple Trim
4. Define a string non-input parameter that is used to match the user's response to the correct sub-assembly; for example, pickFinish. This

parameter operates as an intermediary between the case table and the Boolean parameters.

5. Add a Boolean non-input parameter to the generic part for each of the relevant finishing options; for example,
   - useOakVeneer – to represent the usage of the Oak Veneer material
   - useSolidOak – to represent the usage of the Solid Oak material
   - useSolidOakMaple – to represent the usage of the Solid Oak with Maple Trim material.
6. Add a default expression to each Boolean that evaluates the intermediary parameter; for example:

   pickFinish.equals("OakVeneer") – for the Oak Veneer Boolean

   pickFinish.equals("SolidOak") – for the Solid Oak Boolean

   pickFinish.equals("SolidOakMaple") – for the Solid Oak with Maple Trim Boolean
7. Add the Boolean parameter to the **Inclusion Option** of this generic part for the relevant child part for each option, as follows:
   - useOakVeneer – for the Oak Veneer finish sub-assembly.
   - useSolidOak – for the Solid Oak finish sub-assembly
   - useSolidOakMaple – for the Solid Oak with Maple Trim finish sub-assembly.
8. Establish a case table constraint to map the user's responses to the correct value of the intermediary parameter. For example, consider the case table constraint pickFinish

askFinish	pickFinish
Oak Veneer	OakVeneer
Solid Oak	SolidOak
Solid Oak with Maple Trim	SolidOakMaple
Maple Veneer	MapleVeneer
Solid Maple	SolidMaple
Solid Maple with Oak Trim	SolidMapleOak

Important Note: For this approach to work properly, all values in the pickFinish column of the case table must be unique.

In this approach, after the user selects a particular value, such as Oak Veneer, the case table assigns the value **OakVeneer** to the parameter pickFinish. The Boolean parameter useOakVeneer evaluates its default

expression `{pickFinish.equals("OakVeneer")}` as **true** and the subassembly for the Oak Veneer is included in the variant product structure.

Selecting a Single Option from a Rapidly Changing Set of Options

In some cases, you may wish to select a single option from a set of possible options, but the set of possible options is extremely large or changes rapidly. For example, you may wish to add a company logo or name to a product that you are customizing for a particular environment. In this case, you may not know which company logo files exist in the system at any given time. You only know the company logo file that each product configuration needs.

The technique used in this approach relies on the variant matching capabilities of PDMLink to automatically find, and re-use, the right company logo file.

In this case, the suggested approach is:

1. Create a string attribute such as **Company Name** using the Type Manager.
2. Assign this attribute to every part or part soft-type within PDMLink that contains a company logo file.
3. For each part that contains a company logo, ensure that the string attribute has an appropriate value. For example, the file PTC_logo.jpg might have a **Company Name** attribute with a value of *PTC*.
4. Create a generic part that represents the product or product portion uses the logo file.
5. Add a string input parameter to the generic part; for example, askName.
   – Define an appropriate prompt value for the string parameter such as *Enter the Company Name for this product*.
   – Ensure that this parameter is mapped to the string attribute; for example, **Company Name**.
6. Ensure that each part that contains a company logo is attached as a variant to the generic part that represents the product or product portion.

Note: The variants can be viewed from the **Information Page** of the generic part by selecting **Related Objects** > **Variants**.

Although the product structure does not show any of these possible options, the variant part with the matching **Company Name** attribute is automatically identified and included in the variant product structure when the user requests deliverables for their variant specification.

Common Logic Expressions

Comparing Two String Values

In many cases, you may wish to compare two string values or to compare the value of a string parameter to a particular string.

For example, you may wish to determine if the user entered the value such as Deluxe for the string parameter askSize.

In this case, the suggested approach is:

1. Define an expression for this parameter that evaluates the user's response, such as:

   ```
   askSize.equals("Deluxe")
   ```

If the user enters the value Deluxe, it is stored in the parameter askSize and the expression is evaluated as **true**.

Note: You cannot use a simple equals operator (=) because string values must be evaluated using Java methods, such as *equals*.

In some cases, you may have assigned a string value to a parameter to make it easier to manage multiple comparison expressions in a consistent manner. In this case, the expression would be:

```
askSize.equals(deluxeparameter)
```

where the default value of the string parameter deluxeparameter was defined as Deluxe.

Note: To evaluate a string expression that contains one or more double quotes ("), you must identify, or escape, each double quote with a backslash (\) as shown in the following example:

```
myname.equals("Robert \"Bob\" Smith")
```

Combining Two Expressions with an AND

You may wish to evaluate two responses provided by the user to include a certain component in the variant only if the user's responses are equal to specific values.

For example, you might request the user to specify the size of a table, such as small, medium, or large, and the materials for the table top, such as wood or metal. And there might be a technical requirement that the large table with the metal table top requires an additional support member for safety reasons.

In this case, the suggest approach is to combine two expressions using an AND operator (&&) as follows:

1. Create a generic part.
2. Add two string input parameters to the generic part; for example, askSize and askMaterial.
   - Define an appropriate prompt value for each string parameters such as *Select the desired size* and *Select the desired material*.
   - Define appropriate constraints for each parameter such as small, medium and large for the askSize parameter and wood and metal for the askMaterial parameter.
3. Add a part or structure of parts to the generic part to represent the additional support.

4. Add a Boolean non-input parameter to the generic part; for example:
   – useExtraSupport to represent the usage of the additional support.
5. Define the expression for the Boolean parameter as a combination of the values of the two string parameters; for example:

```
askSize.equals("large") && askMaterial.equals("metal")
```

6. Add the Boolean parameter. For example, add useExtraSupport to the **Inclusion Option** of this generic part for the additional support part or structure of parts.

If the user selects the value **large** for the askSize parameter and the value **metal** for the askMaterial parameter, then the expression is **true** and the additional support part or part structure is included in the variant. If the user provides any other response, then either the askSize or the askMaterial parameter is **false** and the extra support part is excluded from the variant.

Including Two Expressions with an OR

You may wish to evaluate two responses provided by the user to include a certain component in the variant if either of the user's responses is equal to specific values.

For example, you might request the user to specify the size of a table, such as small, medium, or large and the materials for the table top, such as wood or metal. There might also be a technical requirement that the small table or the wood table top required the use of a smaller shipping carton.

In this case, the suggest approach is to combine two expressions using an OR operator (| |) as follows:

1. Create a generic part.
2. Add two string input parameters to the generic part; for example, askSize and askMaterial.
   – Define an appropriate prompt value for each string parameters such as *Select the desired size* and *Select the desired material*.
   – Define appropriate constraints for each parameter such as small, medium and large for the askSize parameter and wood and metal for the askMaterial parameter.
3. Add a part or structure of parts to the generic part to represent the small carton.
4. Add a Boolean non-input parameter to the generic part; for example,
   – useSmallCarton to represent the usage of the small carton.
5. Define the expression for the Boolean parameter as a combination of the values of the two string parameters; for example,

```
askSize.equals("small") | | askMaterial.equals("wood")
```

6. Add the Boolean parameter. For example, add useSmallCarton to the **Inclusion Option** of this generic part for the small carton part or structure of parts.

If the user selects the value **small** for the askSize parameter OR the value **wood** for the askMaterial parameter, then the expression is **true** and the small carton part or part structure is included in the variant. If the user provides any other response, then either the askSize or the askMaterial parameter is **false**, and the small carton part is excluded from the variant.

Making Two Non-String Expressions Equal

You may find it useful to define two parameters or expressions that are equal to one another. For example, you may wish for the speed of the two fans that move air into and out of a chamber to be the same because the chamber is intended to be air-tight.

In this case, the suggested approach is:

1. Create a generic part.
2. Define two, non-string parameters such as parameter1 and parameter2.
3. Define a constraint that establishes equality between the two expressions, such as:

```
parameter1 == parameter2
```

Note: This approach establishes a two-way equality between these two parameters. Therefore, any change to either of the parameters automatically and immediately propagates to the other parameter.

Simple If/Then Expressions using Case Tables

One of the most common logic expressions is the If/Then statement. For example, you might want to ask the user to select a color for the exterior of a product and then automatically select a complimentary interior color. A case table is a set of conditions arranged in rows where each row represents a single If/Then expression.

Therefore, the suggest approach is:

1. Create a generic part.
2. Define a string input parameter; for example, askExteriorColor.
   - Establish a prompt expression such as *Select the desired exterior color*.
   - Establish the valid exterior color values for this parameter's constraint using values such as:
     - Red
     - White
     - Blue

3. Define a second, non-input string parameter; for example, interiorColor.
   - Establish the valid interior color values for this parameter's constraint using values such as:
     - Pink
     - Grey
     - Green
4. Define a case table, for example, pickInteriorColor, to automatically select the interior color based upon the user's exterior color selection, such as:
   - Add both parameters, askExteriorColor and interiorColor, to the case table.
   - Ensure that the case table contains the following values:

askExteriorColor	interiorColor
Red	Pink
White	Grey
Blue	Green

In this case, once the user selects the first parameter, askExteriorColor, the system automatically assigns the second parameter, interiorColor, the value from the case table.

In essence, each row of the case table represents a simple If/Then statement, such as:

```
If askExteriorColor = Red, then interiorColor = Pink
```

Complex If/Then Expressions using Case Tables

In many situations, a simple If/Then expression is not sufficient. For example, you may have a complex set of input conditions and multiple corresponding output values.

In this case, the suggested approach is to establish a case table, as before, but with many more columns, where some of the columns represent the input conditions and other columns represent the output conditions.

Consider the following example:

askMaterial	askTrim	askInlay	supportMaterial	legMaterial
Oak	Walnut	White	Oak	Walnut
Oak	Walnut	Black	Walnut	Oak
Oak	Cherry	White	Oak	Walnut

askMaterial	askTrim	askInlay	supportMaterial	legMaterial
Oak	Cherry	Black	Walnut	Oak
Maple	Cherry	White	Maple	Cherry
Maple	Cherry	Black	Cherry	Maple
Maple	Walnut	White	Maple	Cherry
Maple	Walnut	Black	Walnut	Maple

In this example, a rectangular table has a top with three different material combinations: the material of the top, the material of the trim and the color of the inlay. The first three columns of this case table represent these values.

Based upon these selections, the material for the support and the legs of the table are automatically selected – as shown by the last two columns of this case table.

After the values of askMaterial, askTrim, and askInlay are specified by the user, the case table assigns the corresponding values for supportMaterial and legMaterial.

For example, if the user selects a material of **Maple**, a trim of **Cherry** and a **White Inlay**, the support is defined as **Maple** and the legs are defined as **Cherry**.

Controlling Available Options using Case Tables

Another common situation is when you have two options and you want to control the possible choices for the second option based upon what the user selects for the first option.

Consider the following example:

askMaterial	askTrim
Oak	Cherry
Oak	Walnut
Maple	Cherry
Maple	Walnut
Maple	Oak
Walnut	Cherry

In this case, if the user selects a material of **Oak**, the only valid choices for the trim are **Cherry** and **Walnut**.

You can also use this technique to automatically drive, or define, the value of a related parameter. For example, if the user specifies a value for askMaterial of **Walnut**, then the only valid value for askTrim is **Cherry**.

If the parameter askTrim is defined as *hide when driven* = *true* on its **UI Properties** tab, then the user is not required to provide a value for askTrim if askMaterial is specified as **Walnut**.

Note: remember that parameters are processed by the system on a page-by-page basis; therefore, if you want askTrim to be automatically specified by the system, you must place askTrim on a page after the page where askMaterial is specified.

Conditional Expressions

In certain situations, you may require a parameter to have two different values based upon a certain condition. For example, you have a small table fitted with casters to make it easily moveable; however, for some situations, the table might be fitted with two locking casters so that the movement of the table can be more easily controlled.

While you could achieve this result using a case table, you could also use a conditional expression such as:

```
numberStandardCasters == (useLockingCaster) ? 2 : 4
```

This expression means:

If useLockingCaster is true, then numberStandardCasters is 2, else it is 4

In this example, numberStandardCasters is an integer parameter that is used to define the number of standard, or non-locking, casters on the table. This parameter should be defined using either a type of integer or real number and then assigned to the usage link between the parent part, in this case the small table, and the child part, in this case the standard casters, using the **QuantityOption** field on the **Uses** tab for the parent part.

This example also uses the Boolean parameter useLockingCaster which presumably has a value of **true** if locking casters are desired and a value of **false** when they are not. Therefore, if the locking casters are desired, the value of useLockingCaster is **true** and the value of numberStandardCasters is equal to **2**.

Note: Although a conditional expression of this type is very powerful, it cannot be used to perform conditional assignments. Expressions similar to the following are not supported:

```
numberStandardCasters == (useLockingCaster) ? legLength ==
48 : legLength == 52
```

Creating Modular Generic Product Structures

Designing a Module Product Structure

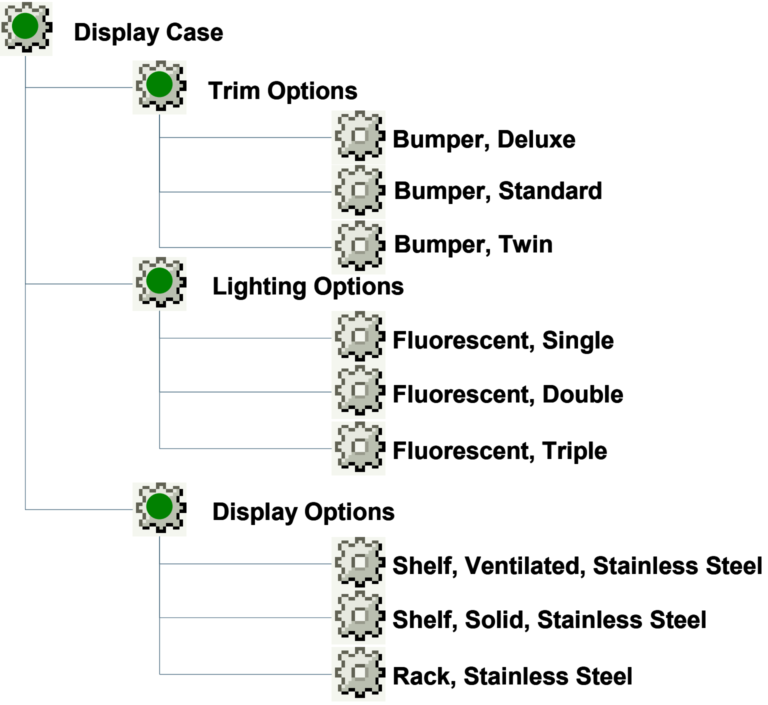
Many products have a number of sub-systems that have their own sets of optional elements. In some cases, the options for each sub-system are

processed independently by the system in a logical progression; however, in other cases, it may be necessary or appropriate for the available options in one sub-system to be affected by selections made in other sub-systems.

For example, consider a metal display case that is available in several different models and with a variety of display, trim, and lighting options. Depending on the display case model that is selected, the options that are available for the display, trim and lighting elements might be different.

For a relatively simple product structure with only a few modules and a limited set of options, it is relatively easy to create and test a suitable generic product structure; however, if the product structure has as few as 5 modules, each with 6 possible options to choose from, the total number of permutations (5 x 6 or 30, in this case) can be overwhelming.

The suggested approach is to organize the generic product structure into generic part option modules attached to a top-level generic part as shown in the following image.



For this example, assume that the display case is available in 5 different models, Model_01 through Model_05, and that some of the options are only applicable to certain models, as shown:

Display Case Model	Trim Options	Lighting Options	Display Options
Model_01	Twin Standard	Single Double	Shelf, Ventilated Shelf, Solid
Model_02	Standard Deluxe	Double, Triple	Shelf, Ventilated Shelf, Solid

Display Case Model	Trim Options	Lighting Options	Display Options
Model_03	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Rack
Model_04	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Shelf, Solid Rack
Model_05	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Shelf, Solid Rack

The next step is to establish the appropriate logic elements in a modular fashion so that each option module can be tested independently and then integrated into the full product structure.

For example, consider the logic for the **Trim Options** module. This module requires 4 parameters as follows:

askModel

- String, Input Parameter
- Prompt = "(TRIM OPTIONS) Select the desired display case model"
- Constraint:"Model_01", "Model_02", "Model_03", "Model_04", "Model_05"
- "hide when driven" = true

useStandard

- Boolean, non-input parameter
- Added to the **Inclusion Option** in the **Uses** tab of the Display Case generic part for the Bumper, Standard part

useTwin

- Boolean, non-input parameter
- Added to the **Inclusion Option** in the **Uses** tab of the Display Case generic part for the Bumper, Twin part

useDeluxe

- Boolean, non-input parameter
- Added to the **Inclusion Option** in the **Uses** tab of the Display Case generic part for the Bumper, Deluxe part

Next, you need a case table such as pickTrim to control the applicability of the trim options for each model of the display case, as follows:

askModel	useStandard	useTwin	useDeluxe
Model_01	yes	yes	no
Model_02	yes	no	yes
Model_03	yes	yes	yes
Model_04	yes	yes	yes
Model_05	yes	yes	yes

These logic expressions allow you to test the **Trim Options** module independently so that you can ensure that everything is working as you intended.

If you use the same approach for each of the modules, though, the user is asked repeatedly what model of Display Case they want, which is undesirable. Therefore, after you create each of the modules and verify that they work correctly, you need to integrate them together using the following approach:

1. Create a suitable parameter to determine the model within the Display Case generic part as follows:

   askModel

   – String, Input Required Parameter

   – Prompt = “Select the desired display case model”

   – Constraint:“Model_01”, “Model_02”, “Model_03”, “Model_04”, “Model_05”

2. Establish an equivalency between askModel in the Display Case generic part and all of its descendents.

3. Remove the constraints for each askModel parameter in each of the option modules.

Note: Removing these constraints is necessary to avoid conflicts between the Display Case top-level generic part and the generic parts that comprise each module.

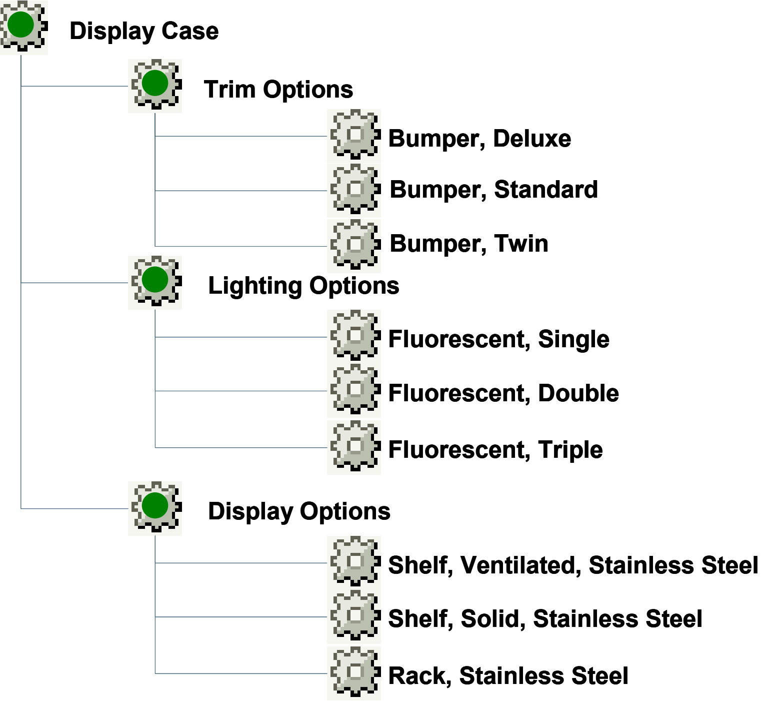
4. Test the completed generic product structure. Because you have already tested each of the modules, you only need to verify that the value of askModel entered by the user in the Display Case top-level generic part is correctly copied to each of the modules.

Tip: If you display the BOM Tree in the Specification Editor and use the **Apply** button, you can quickly see whether the current value of askModel in the Display Case generic part has been copied to each of the modules.

Passing Parameter Values between Sections of a Product Structure Using Equivalencies

One of the most common techniques for passing information between sections of a product structure involves using equivalencies. This technique is most commonly used to pass information from a parent object down to many or all of its child objects.

For example, you might have a Display Case that consists of several different components.



In this example, the Display Case is available in 5 different models, Model_01 through Model_05, and some of the options are only applicable to certain models, as shown:

Display Case Model	Trim Options	Lighting Options	Display Options
Model_01	Twin Standard	Single Double	Shelf, Ventilated Shelf, Solid
Model_02	Standard Deluxe	Double, Triple	Shelf, Ventilated Shelf, Solid
Model_03	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Rack
Model_04	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Shelf, Solid Rack

Display Case Model	Trim Options	Lighting Options	Display Options
Model_05	Twin Standard Deluxe	Single Double Triple	Shelf, Ventilated Shelf, Solid Rack

In situations like this, the user specifies the model of the Display Case and that information needs to be communicated, or passed, between different sections of the product structure.

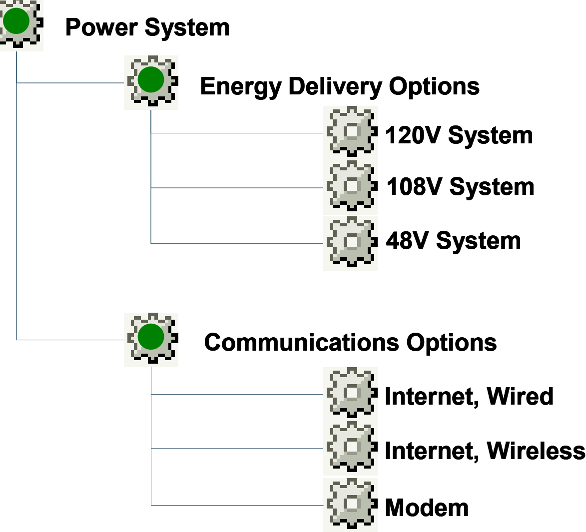
As an example, you might define a parameter such as askModel for the Display Case generic part and then pass the value of this parameter to the generic parts for each of the options.

In this case, because the objective is to share the value of this parameter from the Display Case generic part to a number of child generic parts in the structure, the suggested approach is to define an equivalency for the askModel parameter.

Passing Parameter Values between Sections of a Product Structure Using Reference IDs

Another common technique for passing information between sections of a product structure involves the usage of Reference IDs. This technique is most commonly used when passing information between a parent object and one of its child objects or between a child object and a parent object that are separated by several levels.

For example, you might have a Power System that consists of several different components.



In this example, the Power System is available in three different models with different voltage levels and three different communications options.

In situations like this, the user specifies the desired model of the Power System and that information must be communicated, or passed, to the

Energy Delivery Options section of the product structure but the information is not needed in the Communications Options section.

For example, you might define a parameter such as askModel for the Power System generic part and then pass the value of this parameter to the Energy Delivery Options generic part.

In this case, since the objective is to share the value of the parameter from the Power System generic part only to the Energy Delivery Options child generic part in the structure, the suggested approach involves establishing a Reference ID and a constraint as follows:

1. Create a suitable parameter to determine the model within the Power System generic part as follows:

   askModel

   – String, Input Required Parameter

   – Prompt = “Select the desired display case model”

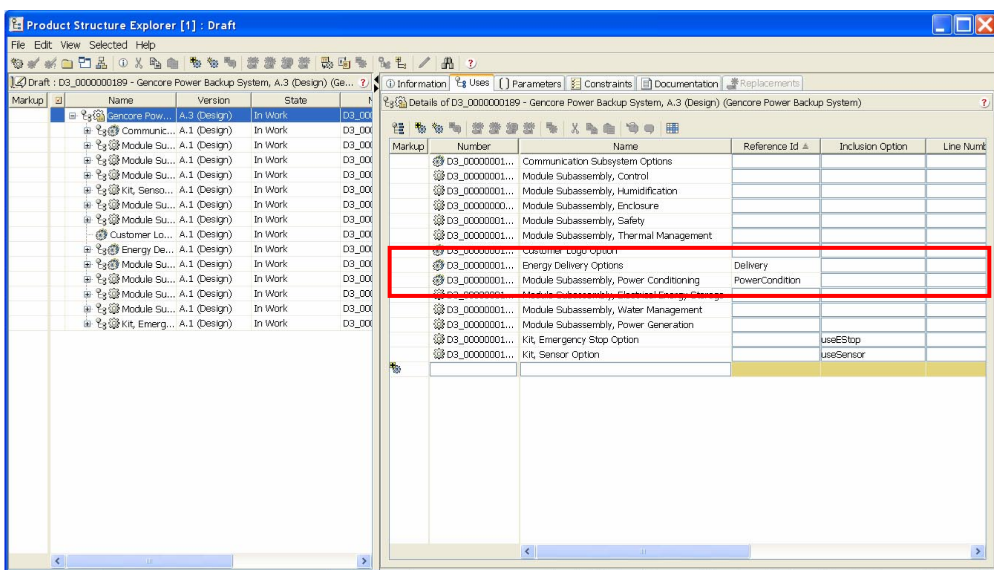
   – Constraint:“120V System”, “108V System”, “48V System”

2. Create a parameter, such as theModel, for the Energy Delivery Option generic part to receive the value of askModel from the Power System generic part.

   theModel

   – String, non-input parameter

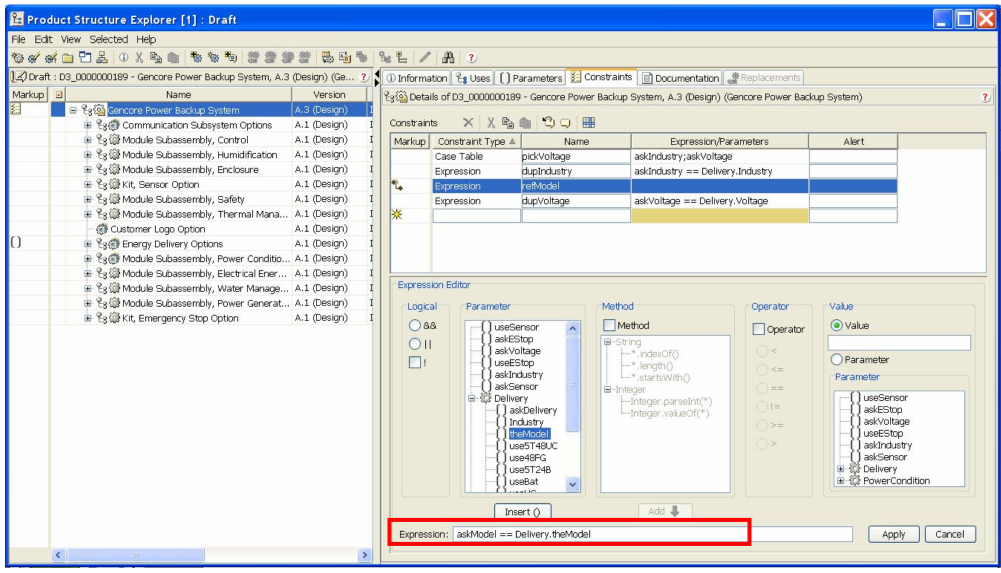
3. Establish a Reference ID, such as Delivery, for the Energy Delivery Option generic part on the **Uses** tab of the Power System generic part as shown in the following image:



4. Create a constraint, such as refModel for the Energy Delivery Option generic part to copy, or pass, the value of askModel from the Power System generic part to theModel parameter for the Energy Delivery Option generic part as follows:

refModel

- Constraint
- Value: askModel == Delivery.theModel



After the user provides a value for askModel in the Power System generic part, its value automatically copies into the theModel parameter for the Energy Delivery Option generic part.

Improving the User Interface of the Configuration Process

Hiding Input Parameters When Users Do Not Have a Choice

In some cases, you may want to define or drive the value of one input parameter based upon the value of another input parameter.

To do this, you need to define a case table to constrain the values of the second parameter.

Consider the following case table example:

askMaterial	askTrim
Oak	Cherry
Walnut	Maple
Cherry	Walnut

In this case, regardless of value selected by the user for askMaterial, only one possible value exists for askTrim. Therefore, as soon as askMaterial has been defined, askTrim is assigned by this case table.

If the parameter askTrim is defined as *hide when driven* = **true** (on its **UI Properties** tab) then the user would not be required to provide a value for askTrim once the value of askMaterial is specified.

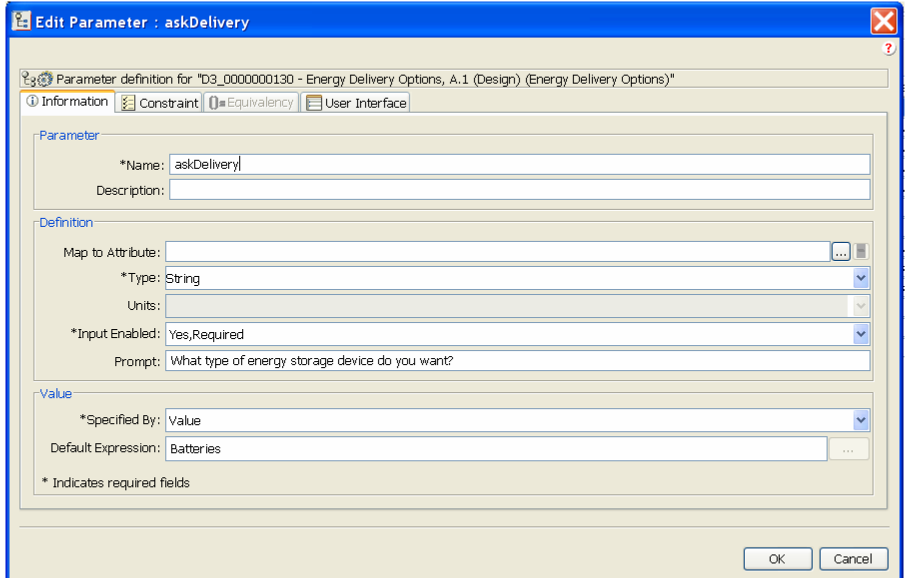
Note: Remember that parameters are processed by the system on a page-by-page basis; therefore, if you want the system to automatically specify the parameter askTrim, you must place askTrim on a page after the page where askMaterial is specified.

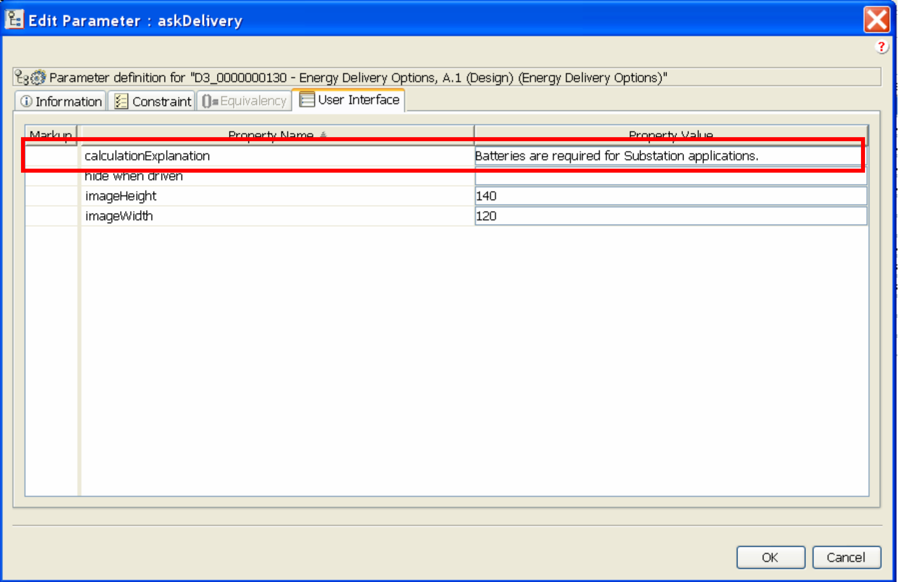
Explaining Input Parameters that Users May Not Specify

In certain situations, the user may select a specific option that automatically eliminates a subsequent option; however, if the same generic product structure can be used in a variety of situations, the user may not understand why certain options can only be selected in certain situations and may become confused or frustrated.

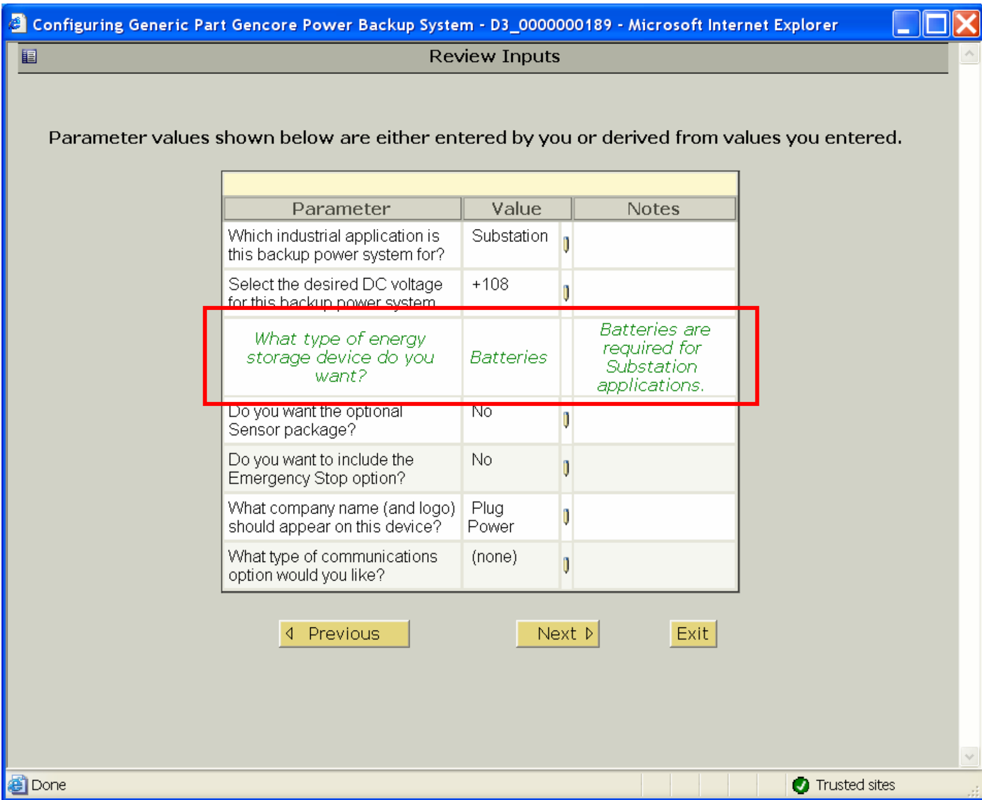
For example, a particular Power Backup System can be used for Telecommunications and Substation applications. The Power Backup System generates electrical power using fuel cells and then stores the power until it is needed. Although the system can store the power it produces in batteries or ultracapacitors, industry regulations require batteries to be used for all substation applications.

To communicate this information to the user, the UI Property **calculationExplanation** can be used, as shown in the following images:





This information is displayed to the user in the Specification Editor as shown in the following image. In this example, the user specified an industrial application of substation which automatically selected the energy storage device of batteries. The **calculationExplanation** text was displayed to the user to explain why this value was automatically selected, as shown.

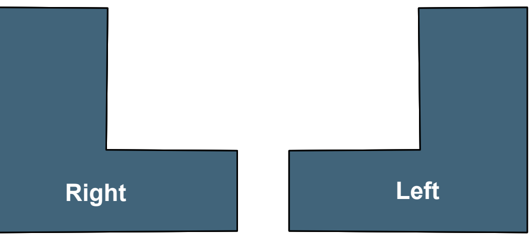


Note: The **calculationExplantion** text is always displayed in the **Review Inputs** screen as shown. It is not possible to dynamically change this text based upon the user's selections.

Adding Images with Parameters with Enumerated Values

Adding appropriate images to a generic product structure can greatly improve the clarity of the information that is presented to the user.

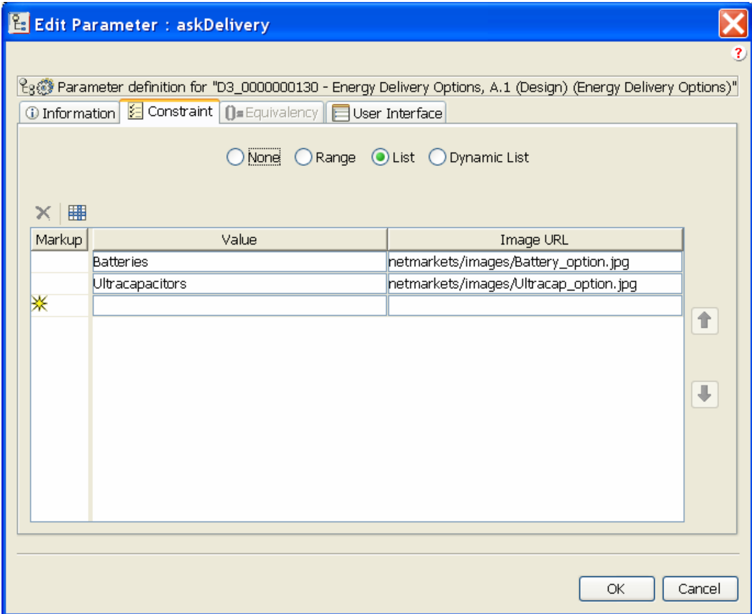
For example, if the user is asked to choose between two L-shaped brackets, one that points to the right and one that points to the left, the user may or may not understand what is meant; however, if the user is provided with two distinct images, the user is more likely to understand the available options.



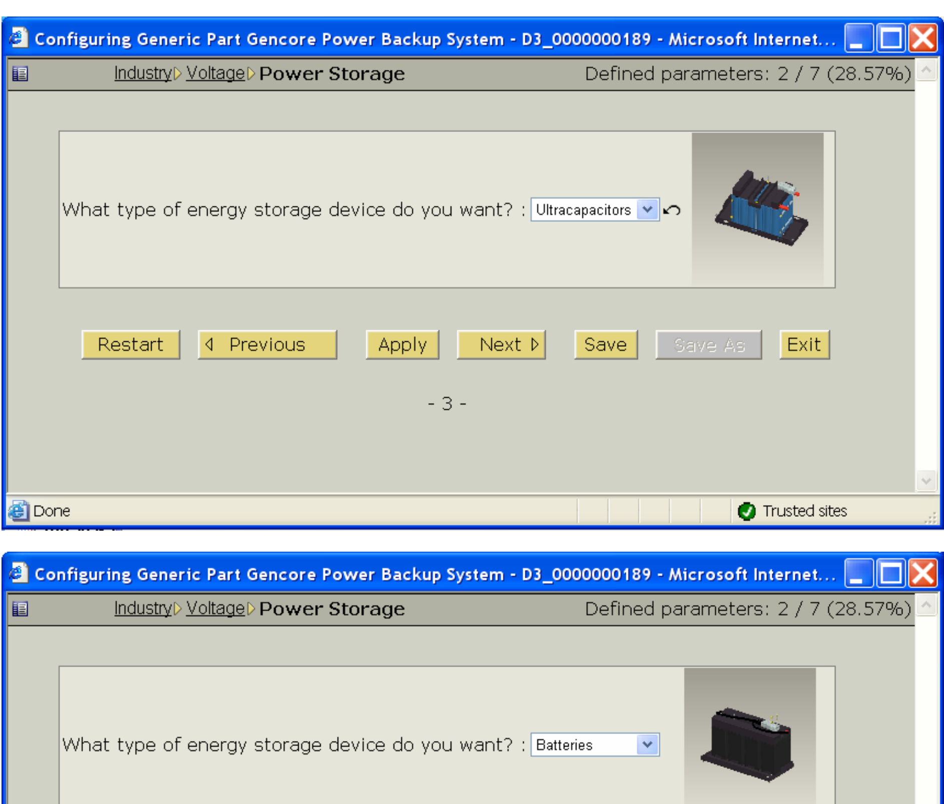
When parameters with enumerated values are defined, you may specify an image URL for each of the values and the Specification Editor displays the image in conjunction with the value to help the user understand the possible values for each option.

As an example, in the Power Backup System generic part, the system generates power using fuel cells and then stores the generated power in either ultracapacitors or batteries.

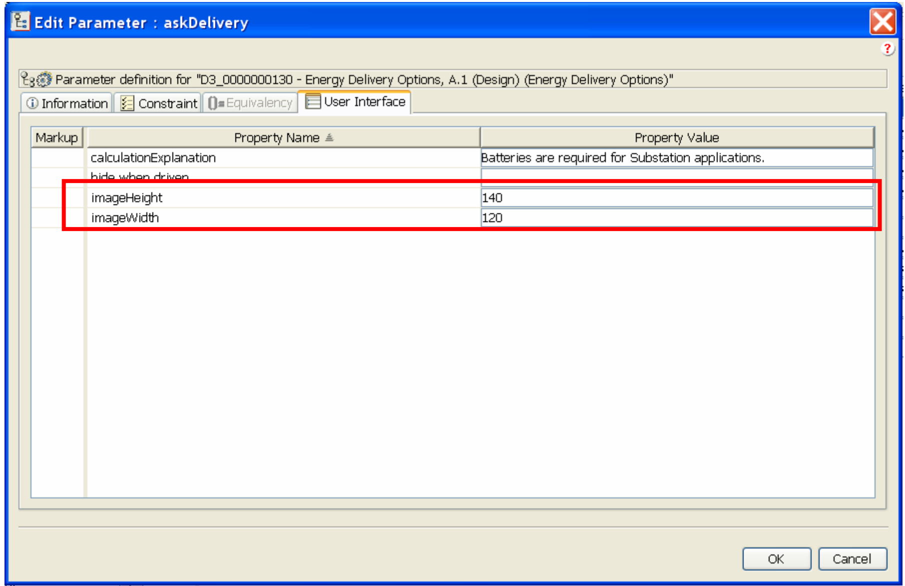
The enumerated values for the parameter askDelivery and the images for each value are specified on the constraints tab for the parameter as shown below.



When the Specification Editor displays this parameter, the corresponding image for each value is automatically displayed, as shown below.

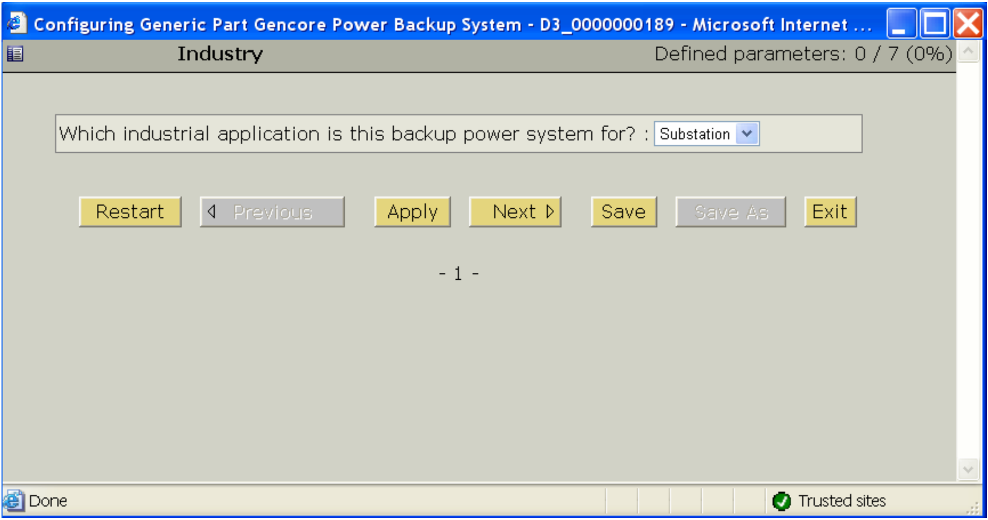


Note: The properties imageHeight and imageWidth should be used to ensure that the size of the image that is displayed in the Specification Editor is appropriate as shown below.

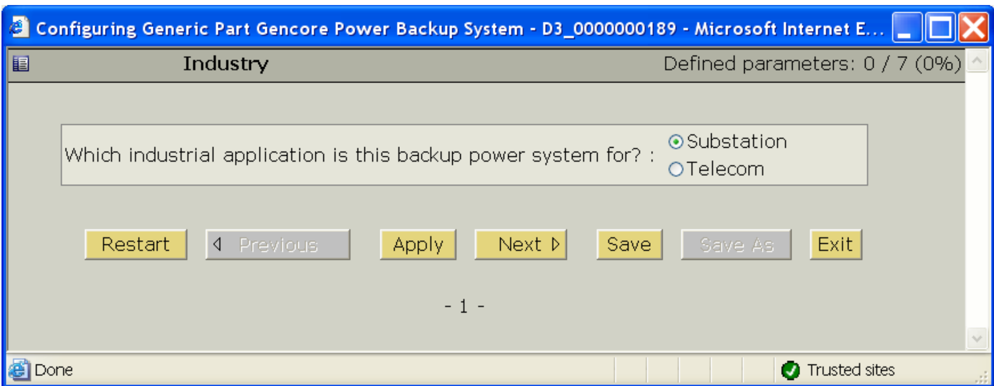


Displaying Values as Radio Buttons

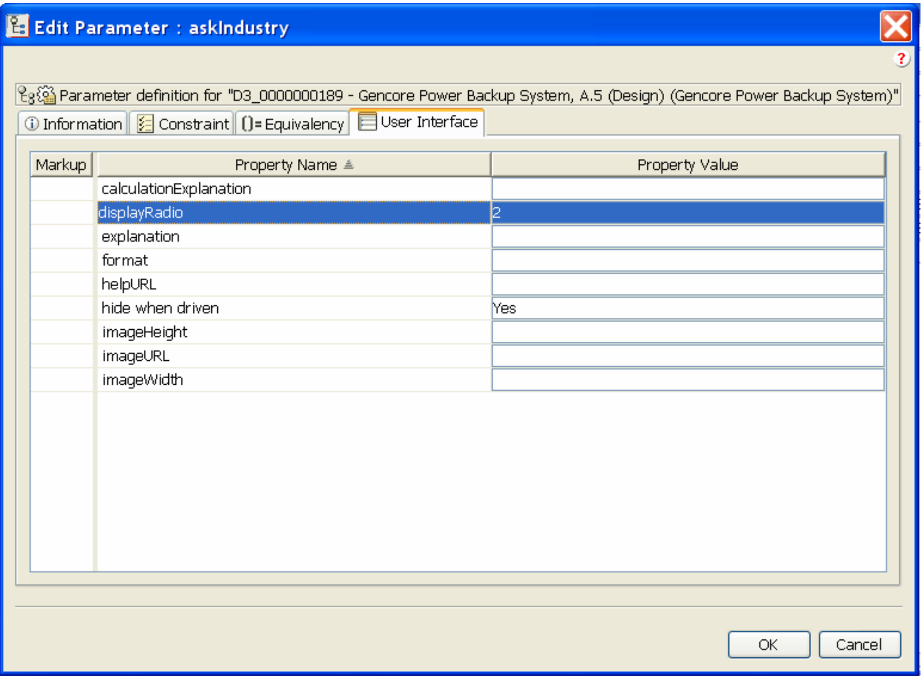
The system automatically displays parameters with enumerated values as drop-down lists as shown in the following image.



In some cases, especially when there are only a few possible values, it may be preferable to display the values as radio buttons as shown in the following image.



To enable this type of display, the UI property **displayRadio** should be set to a numerical value that is equal to or greater than the number of possible values for this parameter, as shown below.



If the number of possible values for the parameter exceeds the value defined for **displayRadio**, the values are displayed using the standard drop-down approach.

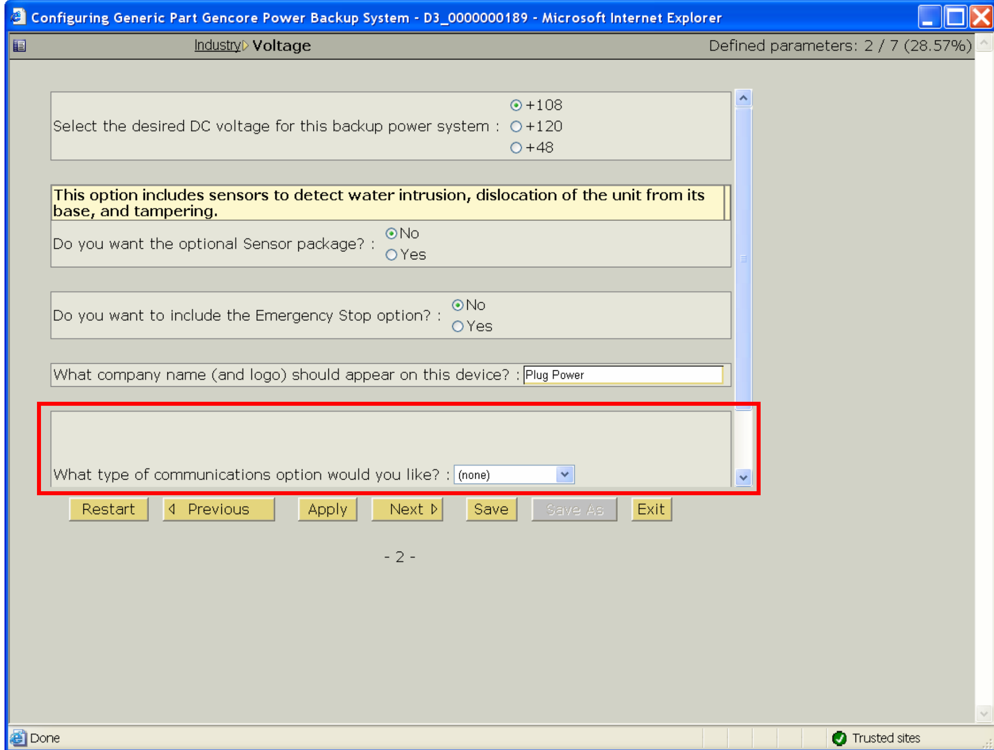
Note: The **displayRadio** property uses the number of possible values for a parameter. Therefore, if you have a parameter with 6 possible values and a case table that constrains the parameter to only display three values at a time, you must define **displayRadio** as 6 if you want to display these three values as radio buttons.

Controlling Input Parameters Using Page Breaks

In almost all situations, it is best to separate the input parameters for a generic product structure into pages. For example, you might want to ensure that the response from one parameter is used to constrain the possible

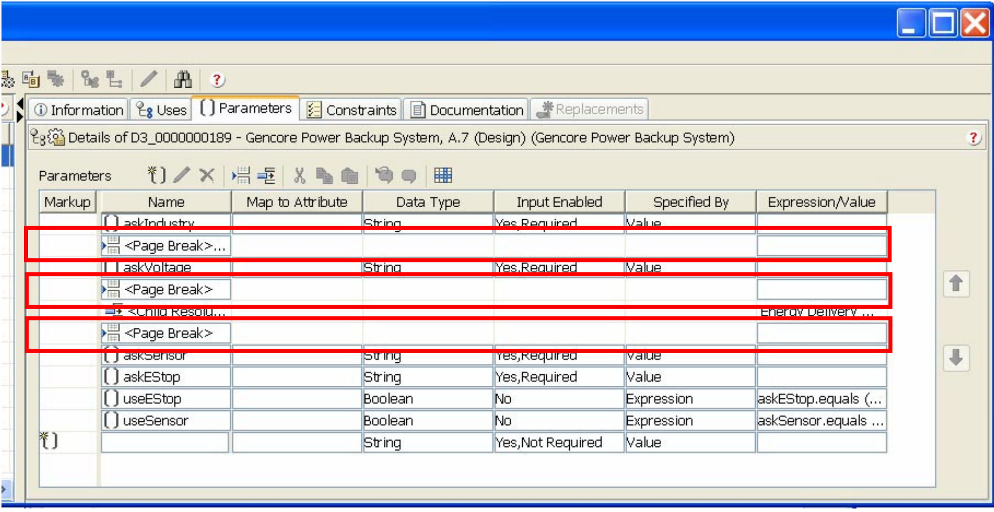
responses for a subsequent parameter, or you might want to limit the number of input parameters that are displayed on a single page.

By default, the system automatically places all input parameters on a single page. Therefore, if you have a large number of input parameters, users must scroll the Specification Editor window vertically to access some of the input parameters, as shown in the following image.



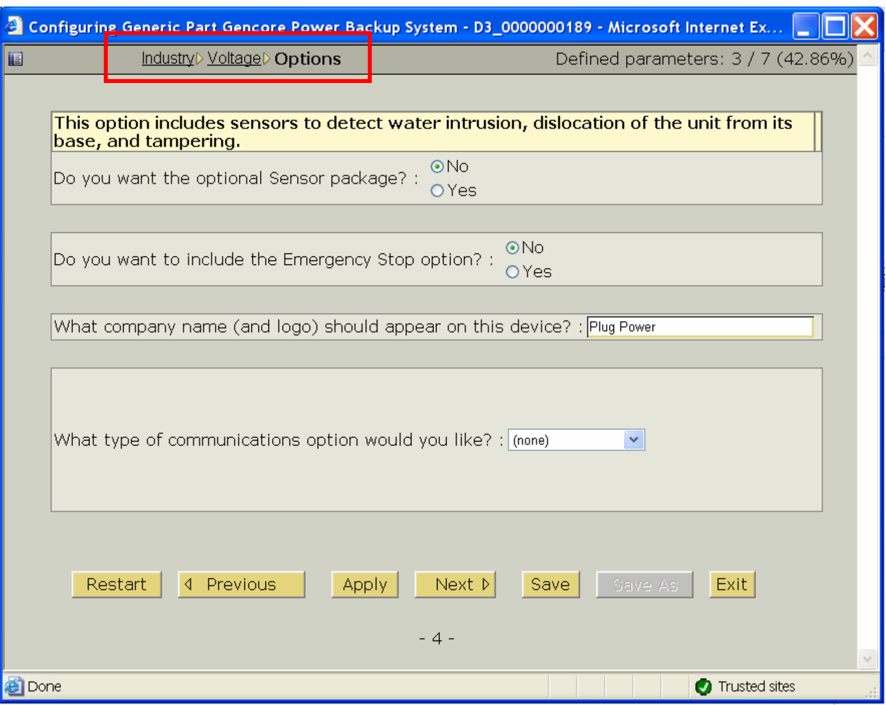
In addition, the Specification Editor processes all parameters on each page as a single operation. So, if you wish to use the value of one parameter to constrain a subsequent parameter, you must divide these two parameters by a page break.

Page breaks are defined in the **Parameters** tab for a generic part as shown in the following image.

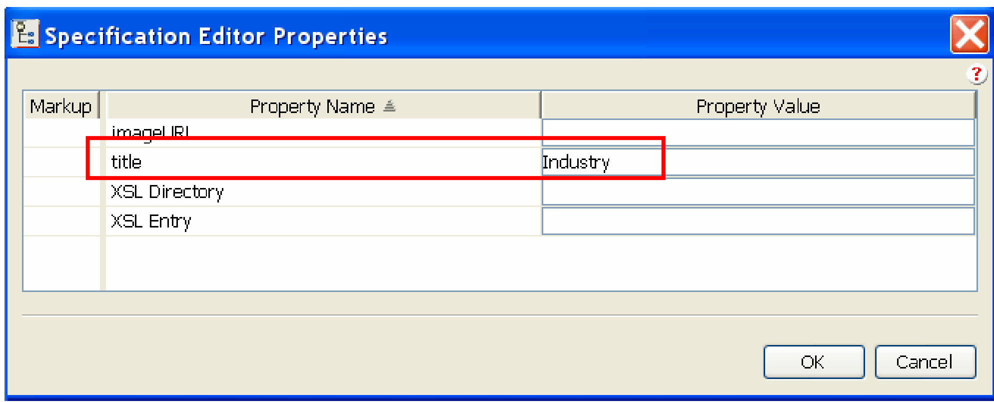


Adding Titles to Specific Pages of Parameters

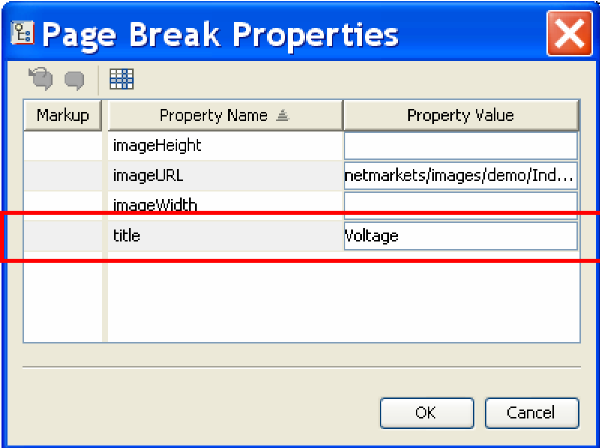
Each page of input parameters that is displayed in the Specification Editor may include a title. These titles are displayed in the Specification Editor as shown in the following image to help organize the input parameters into logical groupings.



You specify the title for the first page of input parameters in the Specification Editor properties as shown below.

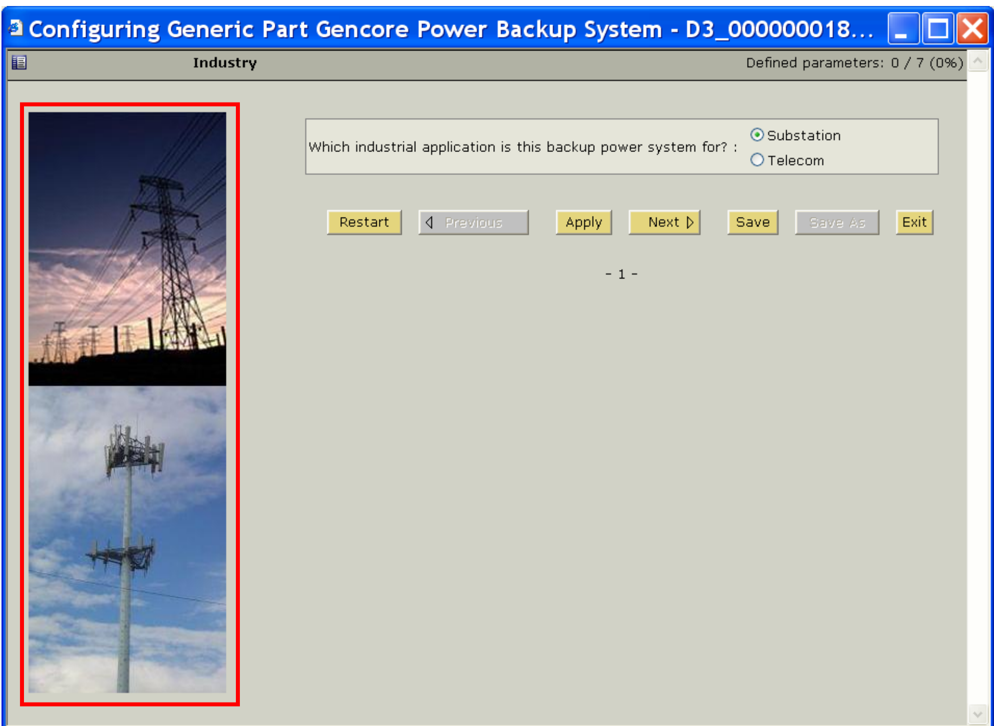


For each successive page of input parameters, you specify the title for each page break as shown below.

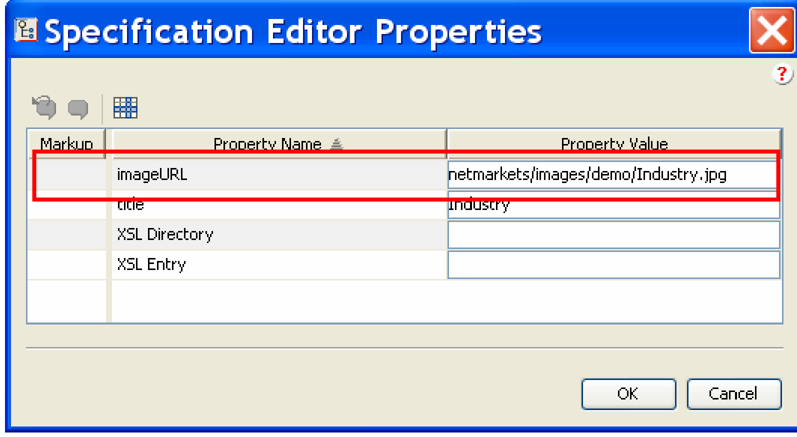


Adding Images for Specific Pages of Parameters

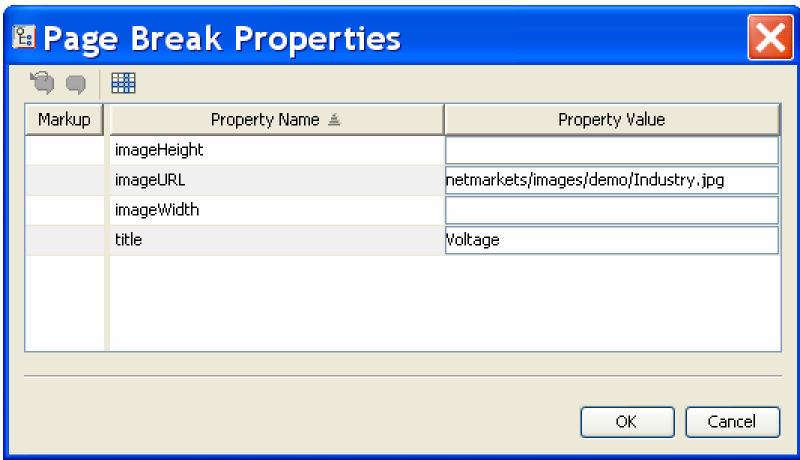
Each page of input parameters that is displayed in the Specification Editor may include an image to help clarify the type of product being configured or to establish branding, as shown in the following image:



To add an image to be displayed on the first page of input parameters, you specify the image URL in the Specification Editor properties as shown below.



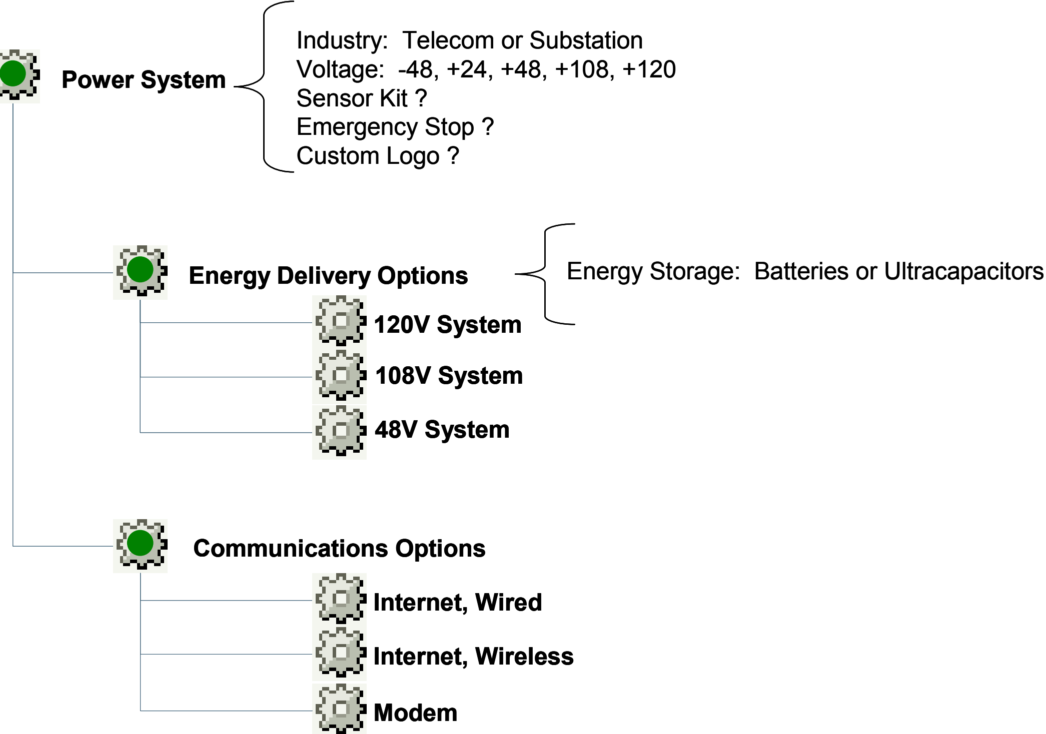
For each successive page of input parameters, you specify the image URL for each page break as shown below.



Controlling the Order of Input Parameters Using Child Resolution

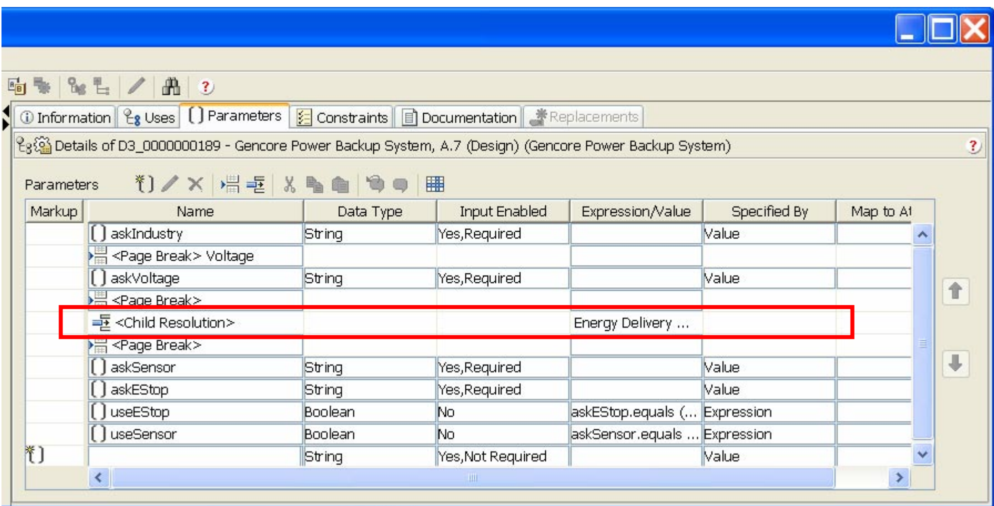
In some situations, you might want one or more input parameters from a child object to be processed in a particular order.

For example, in the following product structure, the user is asked to select the industry and voltage for a Power System, to include or not include a number of options: the Sensor Kit, the Emergency Stop, and the Custom Logo, then to select a Communications Option, and asked to specify whether the energy generated by the system are stored in batteries or ultracapacitors.



Logically, makes more sense to specify the Energy Storage immediately after the Voltage because these two product areas are related.

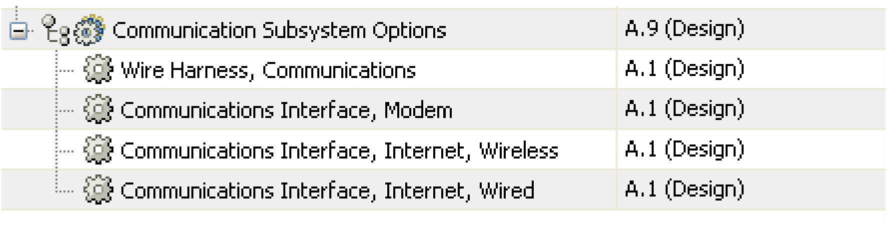
The suggested approach is to insert a **Child Resolution** operation into the **Parameter** tab of the Power System generic part so that the system is instructed to process, or resolve, the input parameters of the child object first, as shown in the following image.



Using Supporting Documents in a Generic Product Structure

Attaching Documents to a Generic Product Structure

In many cases, you might want to attach documents or other supporting information to a generic product structure. For example, each of the Communication Interfaces may have their own installation and usage documentation as shown below.



Since the goal is to ensure that any product variant structure that includes a specific Communications Interface device also includes its corresponding documentation, the suggested approach is to attach the relevant supporting documentation to each device as shown in the following image.

Note: This example illustrates a document that has been modeled as a part.

Communication Subsystem Options	A.9 (Design)
Wire Harness, Communications	A.1 (Design)
Communications Interface, Modem	A.1 (Design)
Manual, Communications Interface, Modem	
Communications Interface, Internet, Wireless	A.1 (Design)
Communications Interface, Internet, Wired	A.1 (Design)

Releasing a Generic Product Structure

Almost all products are designed using an evolutionary approach that allows the first version of the product to be manufactured and sold while the next version of the product is being developed. As a result, PDMLink provides a variety of tools and functions for tracking and managing parts and product throughout their life.

Managing and controlling a generic product structure is very important because an entire family of potential variant part structures can be created from a single generic product structure. In addition, generic product structure may be used by a variety of personnel to rapidly create many, many variant product structures.

As a result, PDMLink provides an additional capability for managing the components and logic that comprise a generic product structure, which is known as a default baseline.

Defining a Default Baseline

Once the design and testing of a generic product structure is completed, it should be reviewed by appropriate personnel as released for use by others.

It is very important that all of the components in the product structure remain synchronized with the logic expressions contained in the various generic parts.

The suggested approach is to:

1. Create a managed baseline using the Product Structure Explorer or the Product Structure Browser that contains the top-most generic part in the structure and all of the other objects in the generic product structure.
2. Promote the top-most generic part to an appropriate release level and use the managed baseline to identify the dependent objects in the generic product structure.

Once the top-most generic part has been promoted, the managed baseline is designated as the default baseline for that revision of that generic part.

Modifying a Default Baseline

Each revision of a top-most generic part may have one and only one default baseline. Therefore, if you create a second managed baseline for the same revision of the top-most generic part, the second managed baseline is

designated as the default baseline for this generic part, replacing the original managed baseline.

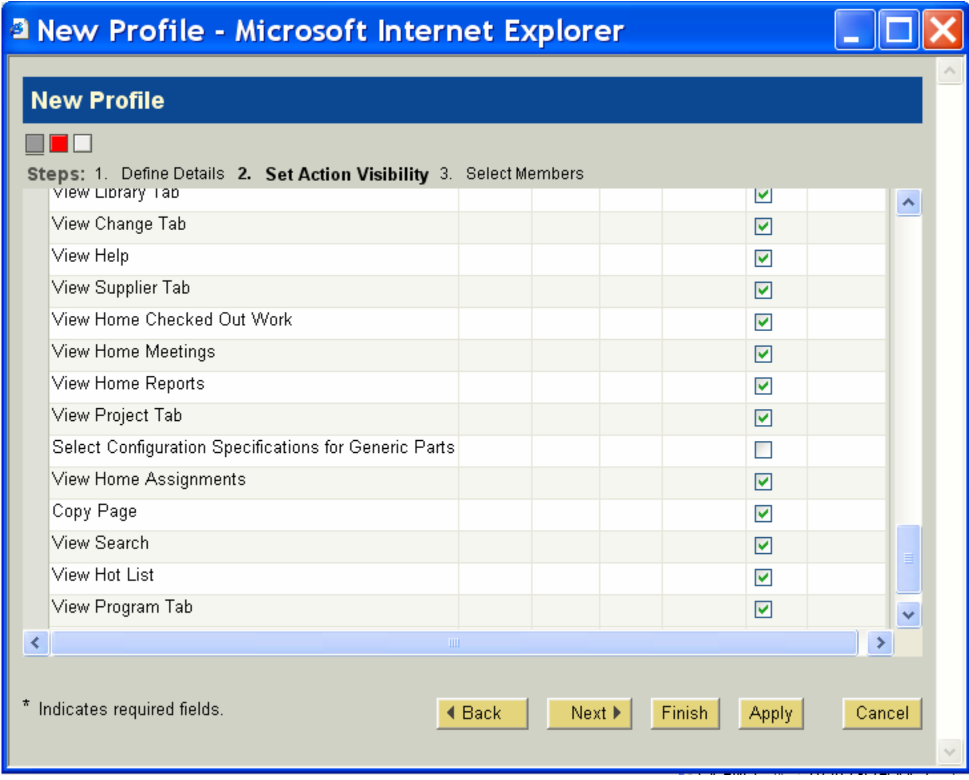
Note: The original managed baseline is retained in PDMLink in its original form except that it is no longer be designated as the default baseline.

Using a Default Baseline

Within PDMLink, there are two broad categories of users: those who are authorized to use and change configuration specifications for generic product structures and those who are not.

The category of users who are not authorized is provided to accommodate those users who are authorized to use approved generic product structures to create variant product structures but who are not authorized to access product components that have not yet been approved. For example, this category of users might include pre-sales support engineers, application engineers, or marketing personnel.

To establish a group of users who are only authorized to use approved generic product structure, create a profile and ensure that the option **Select Configuration Specifications** for generic parts is *not* selected as shown in the following image.



When this option is enabled, the user can use choose any configuration specification for a generic product structure – including the latest available parts. This setting is suggested for users who develop and manage product structures.

When this option is disabled, the user can only use the default baseline for a generic product structure. This setting is suggested for users who are only authorized to create variant product structures from approved generic product structures.

©2007 Parametric Technology Corporation (PTC). The information contained herein is provided for informational use and is subject to change without notice. The only warranties for PTC products and services are set forth in the express warranty statements accompanying such products and services and nothing herein should be construed as constituting an additional warranty. PTC shall not be liable for technical or editorial errors or omissions contained herein. See the Help-About within PTC software products for important information concerning Copyright, Trademarks, Patents and Licensing. PTC, the PTC Logo, The Product Development Company, Pro/ENGINEER, Wildfire, Windchill, Windchill PDMLink, Windchill ProjectLink, Arbortext, Mathcad and all PTC product names and logos are trademarks or registered trademarks of PTC and/or its subsidiaries in the United States and in other countries.